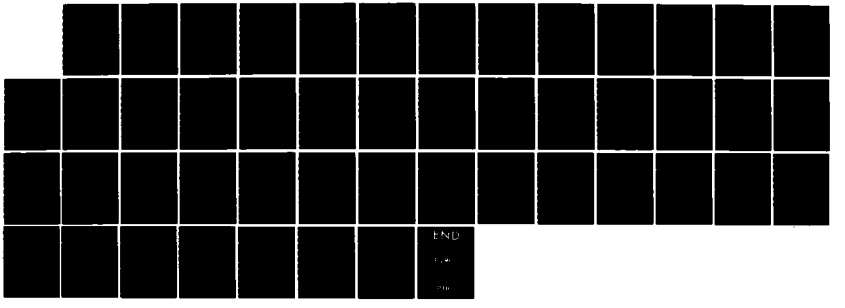
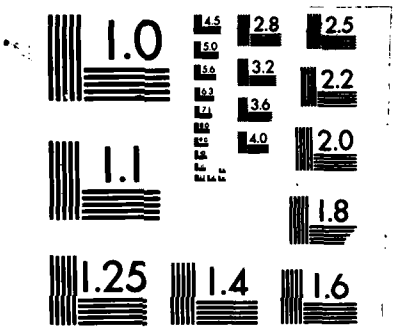
AD-A156 337 INTERPROCESS COMMUNICATION(U) SRI INTERNATIONAL MENLO PARK CA L LAMPORT 11 JUN 85 N00014-84-C-0621 1/1

UNCLASSIFIED F/G 17/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SRI International

(T3)

Interprocess Communication

Final Report for ONR Project N00014-84-C-0621

Leslie Lamport

June 11, 1985

SRI International®

333 Ravenswood Ave. • Menlo Park, CA 94025
415 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Abstract

Interprocess communication is studied without assuming any lower-level communication primitives. Three classes of communication registers are considered, and several constructions are given for implementing one class of register with a weaker class. A formalism is developed for reasoning about concurrent systems that does not assume an atomic grain of action.

Contents

1 Introduction **1**

2 The Constructions **5**

3 The Formal Model **13**
3.1 System Executions 13
3.2 Hierarchical Views 16
3.3 Register Axioms 21
3.4 Systems 25

4 Correctness Proofs for the Constructions **27**
4.1 Proof of Constructions 1, 2, and 3 27
4.2 Proof of Construction 4 29
4.3 Proof of Construction 5 30

5 Conclusion **35**

Appendix **37**

1 Introduction

All communication ultimately involves a communication medium whose state is changed by the sender and observed by the receiver. A sending processor changes the voltage on a wire and a receiving processor observes the voltage change; a speaker changes the vibrational state of the air and a listener senses this change.

Communication acts can be divided into two classes: *transient* and *persistent*. In a transient communication, the medium's state is changed only for the duration of the communication, immediately afterwards reverting to its "normal" state. A message sent on an ethernet modifies the transmission medium's state only while the message is in transit; the altered state of the air lasts only while the speaker is talking. In a persistent communication, the state change remains after the sender has finished its communication. Setting a voltage level on a wire, writing on a blackboard, and raising a flag on a flagpole are all examples of persistent communication.

Transient communication is possible only if the receiver is observing the communication medium while the sender is modifying it. This implies an *a priori* synchronization—the receiver must be waiting for the communication to take place. Communication between truly asynchronous processes must be persistent, the sender changing the state of the medium and the receiver able to sense that change at a later time.

Message passing is often considered to be a form of transient communication between asynchronous processes. However, a closer examination of asynchronous message passing reveals that it involves a persistent communication. Messages are placed in a buffer that is periodically tested by the receiver. Viewed at a low level, message passing is typically accomplished by putting a message in a buffer and setting an interrupt bit that is tested on every machine instruction. The receiving process actually consists of two asynchronous subprocesses: a *main* process that is usually thought of as the receiver, and an *input* process that continuously monitors the communication medium and puts messages in the buffer. The input process is synchronized with the sender (it is a "slave" process) and communicates asynchronously with the main process using the buffer as a medium for persistent communication.

The subject of this paper is asynchronous interprocess communication, so only persistent communication is considered. Moreover, I will restrict myself to unidirectional communication, in which only a single process can modify the state of the medium. With this restriction, two-way communication

requires at least two separate communication media, one modified by each process. However, multiple receivers will be considered. I also restrict my attention to discrete systems, in which the medium has a finite number of distinguishable states. The sender can therefore set the medium to one of a fixed number of persistent states, and the receiver(s) can observe the medium's state.

The form of persistent communication that I have described is more commonly known as a shared register, where the sender and receiver are called the *writer* and *reader*, respectively, and the state of the communication medium is known as the *value* of the register. I will use these in the rest of this paper, so I will consider finite-valued registers with a single writer and one or more readers.

While the practical applications of the algorithms described in this paper will be to "small" registers, the larger purpose is to develop insight into, and formal methods for reasoning about, nonatomic operations to data objects. In the realm of conventional database theory, atomicity is usually called "serializability". Moreover, although the notation used in describing the algorithms suggests a shared-memory implementation, these are really distributed algorithms, since each shared register is modified by only a single process. Thus, the results described here can be regarded as a preliminary investigation of nonserializable operations in a distributed database.

In assuming a single writer, I rule out the possibility of concurrent writes (to the same register). Since a reader only senses the value, there is no reason why a read operation must interfere with another read or write operation. (While reads do interfere with other operations in some forms of memory, such as magnetic core, this interference is an idiosyncrasy of the particular technology rather than an inherent property of reading.) I therefore assume that a read does not affect any other read or any write. However, it is not clear what effect a concurrent write should have on a read.

In concurrent programming, one traditionally assumes that a writer has exclusive access to shared data, making concurrent reading and writing impossible. This assumption is enforced either by requiring the programming language to provide the necessary exclusive access, or by implementing the exclusion with a "readers-writers" protocol [3]. Such an approach requires that a reader must wait while a writer is accessing the register, and vice-versa. Moreover, any method for achieving such exclusive access, whether implemented by the programmer or the compiler, requires a lower-level shared register. At some level, the problem of concurrent access to a shared register must be faced. It is this problem that will be addressed, so I eschew

any approach that requires one process to wait for another.

Asynchronous concurrent access to shared registers is usually considered only at the hardware level, so it is at this level that the methods developed here could have some direct application. However, concurrent access to shared data occurs at high levels of abstraction. One cannot allow any single process exclusive access to the entire social security system's database. While algorithms for implementing a single register cannot be applied to such a database, I hope that the formalism developed for analyzing these algorithms will eventually prove useful for analyzing concurrent systems at a higher level. Nevertheless, it is probably best to think of a register as a low-level component, probably implemented in hardware, when reading this paper.

Hardware implementations of asynchronous communication often make assumptions about the relative speeds of the communicating processes. Such assumptions can lead to simplifications. For example, the problem of constructing an atomic register, discussed below, is shown to be easily solved by assuming that two successive reads of a register cannot be concurrent with a single write. If one knows how long a write can take, a delay can be added between successive reads to ensure that this assumption holds. The results, therefore, apply even to communication between processes of vastly differing speeds.

I therefore make no assumptions about relative process speed and consider a shared register in which a read can overlap (be concurrent with) a write. Three possible assumptions about what can happen when a read overlaps one or more writes are considered.

The weakest possibility is a *safe* register, in which the only assumption made about the value obtained by a read that overlaps a write is that the read obtain one of the possible values of the register—for example, a read of a boolean-valued register must obtain either *true* or *false*. A read that is not concurrent with a write is assumed to obtain the correct value—that is, the most recently written one. However, a read that overlaps a write may return any possible value.

The next stronger possibility is a *regular* register, which is safe (a read not concurrent with a write gets the correct value) and in which a read that overlaps a write obtains either the old or new value. More generally, a read that overlaps any series of writes obtains either the value before the first of the writes or one of the values being written.

The final possibility is an *atomic* register, which is safe and in which reads and writes behave as if they occurred in some definite order. In other

words, for any execution of the system, there is some way of totally ordering the reads and writes so that the values returned by the reads are the same as if the operations had been performed in that order, with no overlapping. (It is also required that this ordering should be a reasonable one; the precise condition is stated below.)

A regular register is obviously stronger than a safe one, since it places a condition on the value returned by a read that overlaps a write. An atomic register is stronger than a regular one because, if two successive reads overlap the same write, then a regular register allows the first read to obtain the new value and the second read the old value. This is forbidden in an atomic register, in which the only allowed possibilities are old-old, new-new, and old-new. In fact, it will be shown that a regular register is atomic if and only if two successive reads that overlap the same write cannot obtain the new then the old value. Thus, a regular register is automatically an atomic one if two successive reads cannot overlap the same write.

These are the only three general classes of register that I have been able to think of. Each class merits study. Safety seems to be the weakest requirement that allows useful communication; I do not know how to achieve any form of interprocess synchronization with a weaker assumption. Regularity asserts that a read returns a "reasonable" value, and seems to be a natural requirement. Atomicity is the most common assumption made about shared registers, and is provided by current multiport computer memories.[1] At a lower level, such as interprocess communication within a single chip, only safe registers are provided; other classes of register must be implemented using safe ones.

Any method of implementing a single-writer register can be classified by three "coordinates" with the following values:

- *safe*, *regular*, or *atomic*, according to the strongest assumption that the register satisfies.
- *boolean* or *multivalued*, according to whether the method produces only boolean registers or registers with any desired number of values.
- *single-reader* or *multireader*, according to whether the method yields registers with only one reader or with any desired number of readers.

[1]However, the standard implementation of a multiport memory does not meet my requirements for an asynchronous register because, if two processes concurrently access a memory cell, one must wait for the other.

This produces twelve classes of implementations, partially ordered by "strength"—for example, a method that produces atomic, multivalued, multireader registers is stronger than one producing regular, multivalued, single-reader registers. In this paper, I address the problem of implementing a register of one class using one or more registers of a weaker class.

The weakest class of register, and therefore the easiest to implement, is a safe, boolean, single-reader one. This seems to be the most natural kind of register to implement with current hardware technology, requiring only that the writer set a voltage level either high or low and that the reader test this level without disturbing it. A series of constructions of stronger registers from weaker ones is presented that allows almost every class of register to be constructed starting from this weakest class. The one exception is that constructing an atomic, multireader register from any weaker one is still an open problem. Most of the constructions are simple; the difficult ones are Construction 4 that implements an m-reader multivalued regular register using m-reader boolean regular registers, and Construction 5 that implements a single-reader multivalued atomic register using single-reader multivalued regular registers.

2 The Constructions

In this section, the algorithms for constructing different classes of registers are described and informally justified. Rigorous correctness proofs are postponed until Section 4, after the necessary formalism is developed.

The algorithms are described by indicating how a write and a read are performed. I will not bother to indicate the initial state of the shared registers—it is the one that would result from writing the initial value starting from any arbitrary state.

The first construction implements a multireader safe or regular register from single-reader ones. It uses the obvious method of having the writer simply maintain a separate copy of the register for each reader. The **for all** statement denotes that its body is executed once for each of the indicated values of i; these separate executions can be done in any order or concurrently.

Construction 1 *Let $v_1, \ldots, v_m$ be single-reader, n-valued registers, where each v_i can be written by the same writer and read by process i, and construct a single n-valued register v in which the operation $v := \mu$ is performed as follows:*

for all i **in** $\{1, \ldots, m\}$
do $v_i := \mu$ **od**

and process i reads v by reading the value of v_i. If the v_i are safe or regular registers, then v is a safe or regular register, respectively.

Any read by process i that does not overlap a write of v does not overlap a write of v_i. If v_i is safe, then this read gets the correct value, which shows that v is safe. If a read of v_i by process i overlaps a write of v_i, then it overlaps the write of the same value to v. It follows easily from this that, if v_i is regular, then v is also regular.

This construction does not make v an atomic register even if the v_i are atomic. If reads by two different processes i and j both overlap the same write, it is possible for i to get the new value and j the old value even though the read by i precedes the read by j—a possibility not allowed by an atomic register.

The next construction is also trivial; it implements an n-bit safe register from n single-bit ones.

Construction 2 *Let $v_1, \ldots, v_n$ be boolean m-reader registers, each written by the same writer and read by the same set of readers. Let v be the 2^n-valued, m-reader register in which the number with binary representation $\mu_1 \ldots \mu_n$ is written by*

for all i **in** $\{1, \ldots, m\}$ **do** $v_i := \mu_i$ **od**

and in which the value is read by reading all the v_i. If each v_i is safe, then v is safe.

The register v is not regular even if the v_i are. A read can return any value if it overlaps a write that changes the register's value from $0 \ldots 0$ to $1 \ldots 1$.

The next construction shows that it is trivial to implement a boolean regular register from a safe boolean register. In a safe register, a read that overlaps a write may get any value, while in a regular register it must get either the old or new value. However, a read of a safe boolean register must obtain either *true* or *false* on any read, so it must return either the old or new value if it overlaps a write that changes the value. A boolean safe register can fail to be regular only if a read that overlaps a write that does not change the value returns the other (wrong) value. To prevent this possibility, one simply does not perform a write that does not change the value.

Construction 3 *Let v be an m-reader boolean register, and let x be a variable internal to the writer (not a shared register) initially equal to the initial value of v. Define v^* to be the m-reader boolean register in which the write operation $v^* := \mu$ is performed as follows:*

```
if x ≠ μ then v := μ;
              x := μ
fi
```

and a read of v^* is performed by reading v. If v is safe then v^* is regular.

There are two known algorithms for implementing a multivalued regular register from boolean ones. The simpler one employs a unary encoding, in which the value μ is denoted by zeros in bits 0 through $\mu - 1$ and a one in bit μ. A reader reads the bits from left to right (0 to n) until it finds a one. To write the value μ, the writer first sets v_μ to one and then sets bits $\mu - 1$ through 1 to zero, writing from right to left. (The idea of implementing shared data by reading and writing its components in different directions was also used in [4].)

Construction 4 *Let $v_1, \ldots, v_n$ be boolean, m-reader registers, and let v be the n-valued, m-reader register in which the operation $v := \mu$ is performed by*

```
v_μ := 1;
for i := μ - 1 step -1 until 1 do v_i := 0 od
```

and a read is performed by:

```
μ := 1;
while v_μ = 0 do μ := μ + 1 od;
return μ
```

If each v_i is regular, then v is regular.

The correctness of this algorithm is not at all obvious. Indeed, it is not even obvious that the **while** loop in the read operation does not "fall off the end" and try to read the nonexistent register v_{n+1}. This can't happen because, whenever the writer writes a zero, there is a one to the right of it. (Since I am assuming that an initial value has been written, some v_i initially equals one.) As an exercise, the reader of this paper can convince himself that, whenever a reading process sees a one, it was written by either

a concurrent write or by the most recent preceding one, so v is regular. The formal proof is given in Section 4.

The value of v_n is only set to one, never to zero. It can, therefore, be eliminated; the writer simply never writes it and the reader assumes its value is one instead of reading it. I will not bother writing down this modification.

Even if all the v_i are atomic, Construction 4 does not produce an atomic register. To see this, suppose that the register initially has the value 3, so $v_1 = v_2 = 0$ and $v_3 = 1$, the writer first writes the value 1 then the value 2, and there are two successive read operations. This can produce the following sequence of actions:

- the first read finds $v_1 = 0$
- the first write sets $v_1 := 1$
- the second write sets $v_2 := 1$
- the first read finds $v_2 = 1$ and returns the value 2
- the second read finds $v_1 = 1$ and returns the value 1.

In this scenario, the first read obtains a newer value (the one written by the second write) than the second read (which obtains the one written by the first write), even though it precedes the second read. This shows that the register is not atomic.

Construction 4 uses $n - 1$ boolean regular registers to make an n-valued one, so it is practical only for small values of n. We would like an algorithm that requires $O(\log n)$ boolean registers to construct an n-valued register. The second method for constructing a regular multivalued register uses an algorithm of Peterson [11] that implements an m-reader n-valued atomic register with $m + 2$ safe m-reader registers; $2m$ atomic boolean 2-reader registers, and two atomic boolean m-reader registers. There is no known algorithm for constructing multivalued m-reader atomic registers from simpler ones. However, we can apply Peterson's algorithm to construct an n-valued single-reader atomic register using three safe single-reader n-valued registers and four single-reader atomic boolean registers. The safe registers can be implemented with Construction 2, and the atomic boolean registers can be implemented with Construction 5 below. Since an atomic register is regular, Construction 1 can then be used to make an m-reader n-valued regular register from $O(3m \log n)$ single-reader boolean regular registers.

Before giving the algorithm for constructing a two-reader atomic register, I prove a result that indicates why no trivial algorithm will work. It asserts that there can be no algorithm in which the writer only writes and the reader only reads; any algorithm must involve two-way communication between the reader and the writer.

Theorem: *There exists no algorithm to implement an atomic register using only a finite number of regular registers that can be written by the writer (of the atomic register).*

Proof: I assume such an algorithm and derive a contradiction. Without loss of generality, I can assume that there is only a single regular register v written by the writer and read by the reader. (Any algorithm that works with multiple registers must also work when those registers are combined into a single large regular register.)

Let v^* denote the atomic register that is being implemented. Suppose that the writer performs an infinite number of writes that change the value of v^*. There must be some pair of values assumed by v^*, call them 0 and 1, such that there are an infinite number of writes that change v^*'s value from 0 to 1. Since v can assume only a finite number of values (the hypothesis states that the original algorithm has only a finite number of registers, and all registers are taken to have only a finite number of possible values), there must exist values $v_0, \ldots, v_n$ of v such that v_0 is the final value of v after each one of an infinite number of writes of 0 to v^*, v_n is the final value of v after each one of an infinite number of writes of 1 to v^*, and, for each $i < n$, the value of v is changed from v_i to v_{i+1} during infinitely many writes that change the value of v^* from 0 to 1.

A read of v^* may involve several reads of v. However, by considering only scenarios in which each of those reads of v obtains the same value, we may assume that each read of v^* reads v only once. Since v assumes each value v_i infinitely often, it must be possible for a sequence of $n + 1$ consecutive reads to obtain the values $v_n, v_{n-1}, \ldots, v_1$.

The read that finds v equal to v_i and the subsequent read that finds v equal to v_{i-1} could both have overlapped the same write of v, which could have been a write that occurred in the process of changing v^*'s value from 0 to 1. Therefore, if the read of v^* that finds v equal to v_i returns the value 1, then the subsequent read that finds v equal to v_{i-1} must also return the value 1, since both reads could be overlapping the same write and, in that case, two successive reads of an atomic register cannot return first the new value, then the old.

The first read, which finds v equal to v_n, must return the value 1, since it could have occurred after the completion of a write of 1. By induction, this implies that the last read, which found v equal to v_0, must return the value 1. However, this read could have occurred after a write of 0 and before any subsequent write, so returning the value 1 would violate the assumption that the register v^* is safe. (An atomic register is *a fortiori* safe.) This is the required contradiction. ∎

This theorem could be expressed and proved using the formalism developed below, but doing so would lead to no new insight. The formal proof of this theorem is therefore left as an exercise for the compulsive reader.

The theorem is false if no bound is placed on the number of values a register can hold. Given a regular register v that can assume an unbounded number of values, an atomic register v^* is implemented as follows. The writer sets v equal to a pair consisting of the value of v^* and a sequential version number. The reader reads v and compares the version number with the previous one it read. If the new version number is higher, then it uses the value it just read; if the new version number is lower, then it forgets the value and version number it just read and uses the previously read value. The correctness of this algorithm follows easily from Proposition 9 of Section 3.3. By assuming registers hold only a bounded set of values, I am disallowing such algorithms.

Finally, we come to the algorithm for constructing a single-reader atomic register from regular ones. To begin, we try to implement an atomic register v^* with a regular register v that holds a pair of values, both normally equal. When v is changed from (ν, ν) (denoting $v^* = \nu$) to (μ, μ) (denoting $v^* = \mu$), it is first set to the intermediate value (ν, μ). The reader reads v and returns the first component unless it obtains (ν, μ) after having returned the value μ the last time, in which case it must return the value μ to avoid a "new-old" sequence.

The preceding theorem shows that this idea, by itself, is not enough. The reader is in a quandary if three successive reads of v obtain the values (μ, μ), (ν, μ), and (ν, ν). The first read simply returns μ; as I just observed, the second read must also return μ; but what can the third read return? The second and third reads could both have overlapped a single write that is changing the value from ν to μ, so returning ν would produce a new-old sequence. On the other hand, the third read could have seen a completely new value, written long after the write that overlapped the second read, so re-

turning μ could violate safety—the requirement that a read not overlapping any write return the correct value.

To overcome this problem, I add another bit to v, which I will call the *color* value. When the reader reads v, it sets a shared one-bit register cr to v's color value. The writer first reads the register cr and sets v to the opposite color. (Thus, the reader tries to make cr and v's color the same, and the writer tries to make them different.) The reader interprets (ν, μ) as a μ only if its previous read saw a μ of the same color. The only source of embarrassment is now if three successive reads return values (μ, μ), (ν, μ), and (ν, ν) that are all the same color. It will be shown in Section 4 that this can happen only if the last read actually overlaps the write of (ν, μ), so it is allowed to return the value μ without violating the safety requirement.

In the following construction, the variable cr is written by the reader and read by both the reader and the writer. A two-reader register is not needed, since the reader can maintain a local variable containing the value that it last wrote into cr. (This is just Construction 1 with $m = 2$ and the writer being the second reader.) Such a local variable would complicate the description, so it is omitted. In the reader's program, the primed variables denote the values read the previous time, except that, if the reader reads (μ, μ) then (ν, μ), both with the same color, then it "forgets about" the latter value.

Construction 5 *Let $\mathcal{V}$ be an n-element set; let w and r be processes; let v, cw denote a single $2n^2$-valued register that can be written by w and read by r, where v has a value in $\mathcal{V} \times \mathcal{V}$ and cw is boolean valued; and let cr be a boolean register that can be written by r and read by w. Define the n-valued register v^*, with values in $\mathcal{V}$, written by w and read by r by letting the write $v^* := \mu$ be performed by:*

$$v, cw := (v_1, \mu), \neg cr;$$
$$v, cw := (\mu, \mu), cw$$

and letting the read operation be performed by the program of Figure 1, where x and x' are local variables with values in $\mathcal{V} \times \mathcal{V}$, cr' is a boolean-valued local variable, and rtn is a local variable with values in $\mathcal{V}$ whose final value is the one returned by the read. Initially, x', cr' equals $(v, cw)[0]$.

```
x, cr := v, cw;
if cr = cr'
   then if x1 = x2
           then if x1 = x1' ≠ x2' ∧ rtn = x2'
                   then skip
                   else x' := x;
                        rtn := x1
                fi
           else if (x = x' ∧ rtn = x2) ∨ x1' = x2' = x2
                   then x' := x;
                        rtn := x2
                   else x' := x;
                        rtn := x1
                fi
        fi
   else x', cr' := x, cr;
        rtn := x1
fi
```

Figure 1: Construction 5: the reader's algorithm.

3 The Formal Model

3.1 System Executions

Almost all models of concurrent processes are based upon indivisible atomic actions as their primitive elements. For example, models in which a process is represented by a sequence or "trace" [1,12,13] assume that each element in the sequence represents an indivisible action. Net models [2] and related formalisms [9,10] assume that the firing of an individual transition is atomic. Operations to a nonatomic shared register cannot be modeled as atomic actions, since these formalisms have no concept of two atomic actions overlapping in time.

One can model a single read or write operation with two atomic actions: a *start* and a *finish* action. I will employ such a model to motivate the formalism. However, in the general view of physical systems based upon special relativity that is discussed in two of my works [7,5], there may be no single real event that precedes all other events in the operation, and no single event that follows all others. I will show that assuming such fictitious *start* and *finish* events would result in no loss of generality. However, it turns out to be easier to reason directly in terms of the nonatomic actions than to use starting and finishing events.

I therefore eschew more conventional formalisms in favor of one introduced in [6] and refined in [5], in which the primitive elements are *operation executions* that are not assumed to be atomic. In this formalism, an execution of a system is represented as a triple $S, \longrightarrow, \dashrightarrow$, where S is a finite or countably infinite set of operation executions, and $\longrightarrow$ and $\dashrightarrow$ are precedence relations on S.

The most general way of viewing the formalism is to consider an operation execution to be a set of points in four-dimensional space-time. Such a view is provided in [5]. While using the same formalism as [5], I will employ a less general but more intuitive model. In this model, an operation execution A is thought of as an activity performed during some time interval $[s_A, f_A]$, where the real numbers s_A and f_A are the starting and finishing times of A. I assume that, at any time, only a finite number of operation executions have begun. Stated formally, a model consists of a set S of operation executions, together with real-valued functions s and f on S such that the following conditions hold for all A and B in S (where I write s_A and f_A instead of $s(A)$ and $f(A)$):

M1. $s_A \leq f_A$

M2. for any real number t: $\{A : s_A < t\}$ is finite

An operation execution A is said to be *instantaneous* if, for any $B \neq A$, the numbers s_B and f_B lie outside the interval $[s_A, f_A]$. Thus, A is instantaneous if and only if we can set s_A equal to f_A (shrinking the interval to a point) without changing the relative order of any starting and finishing times.

Given such a model, we can define the relations $\longrightarrow$ and $\dashrightarrow$ as follows:

$$\begin{aligned} A \longrightarrow B &\equiv f_A < s_B \\ A \dashrightarrow B &\equiv s_A \leq f_B \end{aligned} \tag{1}$$

Thus, $A \longrightarrow B$ means that A finishes before B starts, and $A \dashrightarrow B$ means that A starts no later than B finishes. We read $A \longrightarrow B$ as "A precedes B" and $A \dashrightarrow B$ as "A can affect B".

M1, M2 and (1) imply that the following hold for all operation executions A, B, C, and D in S:

A1. The relation $\longrightarrow$ is an irreflexive partial ordering.

A2. If $A \longrightarrow B$ then $A \dashrightarrow B$ and $B \not\dashrightarrow A$.

A3. If $A \longrightarrow B \dashrightarrow C$ or $A \dashrightarrow B \longrightarrow C$ then $A \dashrightarrow C$.

A4. If $A \longrightarrow B \dashrightarrow C \longrightarrow D$ then $A \longrightarrow D$.

A5. For any A, the set of all B such that $A \not\longrightarrow B$ is finite.

Instead of basing the formalism on this model, I adopt the more general view of [5] and take A1–A5 as axioms.

Definition 1 *A* system execution *is a triple $S, \longrightarrow, \dashrightarrow$ such that S is a finite or countably infinite set and $\longrightarrow$ and $\dashrightarrow$ are relations on S that satisfy A1–A5.*

Observe that A1 and A4 imply that if $A \longrightarrow B$ and $A \dashrightarrow B$ then $B \not\dashrightarrow A$, so the "and $B \not\dashrightarrow A$" in A2 is superfluous.

Definition 1 differs from the definition of a system execution given in [5] because I am considering only terminating operations. In the more general formalism, Axiom A5 needs the hypothesis that A terminates.

Definition 2 *A* global-time model *of a system execution $S, \longrightarrow, \dashrightarrow$ consists of a pair s, f of real-valued functions on S satisfying M1, M2 and (1). It is said to be* nondegenerate *if, for all A: $s_A < f_A$ and for all $B \neq A$: $s_A \neq s_B$ and $s_A \neq f_B$.*

A nondegenerate global-time model is one in which no two starting or stopping times are identical. The following result states that any global-time model can be turned into a nondegenerate one by tiny perturbations of the starting and finishing times of operation executions. Such perturbations should be allowed, since no physically meaningful result could depend upon completely accurate knowledge of these times. (It makes no physical sense to specify the starting and finishing times of an operation execution down to the fraction of a micropicosecond.)

Proposition 1 *For any any global-time model s, f of a system execution S, $\longrightarrow$,- - → and any $\epsilon > 0$, there exists a nondegenerate global-time model s', f' of S,$\longrightarrow$,- - → such that $|s'_A - s_A| < \epsilon$ and $|f'_A - f_A| < \epsilon$ for all $A \in S$.*

The proofs of this and all other propositions stated in this section are given in the appendix.

In a global-time model, the starting and finishing times of operations are totally ordered. Given two operation executions A and B, s_B must be either greater than or not greater than f_A, so the following condition holds.

A#. For any operation executions A and B with $A \neq B$: $A \longrightarrow B$ or B - - → A.

This condition does not hold for all system executions. (Trivial counterexamples are obtained by noting that the empty precedence relations make any set a system execution.) Condition A# holds only if there is a global-time model.

Proposition 2 *A system execution S,$\longrightarrow$,- - → has a global-time model if and only if A# holds.*

In the more general interpretation of operation executions given in [5], condition A# fails to hold for a pair of operation executions A, B if A and B occur at spatially separated locations, and they both happen within a time interval that is less than the time needed for light to travel between their locations. In most systems of practical interest, A# holds for almost all pairs A, B of operation executions.

The following result shows that we can get a global-time model by adding extra precedence relations.

Proposition 3 *Given any system execution S,$\longrightarrow$,- - →, there exist extensions $\xrightarrow{!}$ of $\longrightarrow$ and - $\overset{!}{-}$ → of - - → such that S,$\longrightarrow$,- $\overset{!}{-}$ → is a system execution satisfying A#.*

Later, I will indicate why we can consider the system execution $S, \longrightarrow, \overset{!}{\dashrightarrow}$ to be a reasonable way of viewing the system execution $S, \longrightarrow, \dashrightarrow$.

A system execution satisfying A# is maximal in the sense that no additional $\longrightarrow$ or $\dashrightarrow$ relations can be added. This is because, for any pair of distinct operation executions A and B, A# implies that either $A \longrightarrow B$, or $B \longrightarrow A$, or $A \dashrightarrow B$ and $B \dashrightarrow A$. In any of these three cases, adding an additional precedence relation would violate A1 or A2.

When trying to understand an algorithm or its correctness proof, it is useful to think in terms of a global-time model, drawing pictures of reads and writes as time intervals. However, I find that the best way to formalize the proof is to use Axioms A1–A5. The additional assumption A#, implicitly introduced when using a global-time model, is not needed.

3.2 Hierarchical Views

The same system can be viewed at different levels of detail, with different operation executions at each level. Viewed at the customer's level, a banking system has operation executions such as *deposit* \$10. Viewed at the programmer's level, this same system executes operations such as $dep\_amt[cust] := 1000$. The fundamental problem of system building is to implement one system (like a banking system) as a higher-level view of another system (like a Pascal program).

A higher-level operation consists of a set of lower-level operations—the set of operations that implement it. Let $S, \longrightarrow, \dashrightarrow$ be a system execution and let $\mathcal{H}$ be a set whose elements, called *higher-level operation executions*, are sets of operation executions from S. We consider the starting time $s^\bullet_H$ of a higher-level operation execution H to be the earliest starting time of all the operation executions it contains, and its finishing time $f^\bullet_H$ to be their latest finishing time. In other words, for every H in $\mathcal{H}$:

$$\begin{aligned} s^\bullet_H &= \min\{s_A : A \in H\} \\ f^\bullet_H &= \max\{f_A : A \in H\} \end{aligned} \tag{2}$$

In order for this to define real-valued functions $s^\bullet$ and $f^\bullet$ on $\mathcal{H}$ that satisfy M1 and M2, it is sufficient for $\mathcal{H}$ to satisfy the following two conditions:

H1. Each element of $\mathcal{H}$ is a finite, nonempty set of elements of S.

H2. Each element of S belongs to a finite, nonzero number of elements of $\mathcal{H}$.

A set $\mathcal{H}$ of subsets of S satisfying H1 and H2 is called a *higher-level view* of S. In most cases of interest, $\mathcal{H}$ is a partition of S, so each element of S belongs to exactly one element of $\mathcal{H}$. However, I allow the more general case in which a single lower-level operation execution is viewed as part of the implementation of more than one higher-level one.

Let $S, \longrightarrow, \dashrightarrow$ be a system execution with a global-time model s, f, and let $\mathcal{H}$ be a higher-level view of S. We can define $s^\bullet$ and $f^\bullet$ by (2) and then use (1) to define $\xrightarrow{\bullet}$ and $\dashrightarrow^\bullet$, obtaining a system execution $\mathcal{H}, \xrightarrow{\bullet}, \dashrightarrow^\bullet$ having $s^\bullet, f^\bullet$ as a global-time model. The precedence relations $\xrightarrow{\bullet}$ and $\dashrightarrow^\bullet$ can be obtained directly from $\longrightarrow$ and $\dashrightarrow$ as follows:

$$\begin{aligned} G \xrightarrow{\bullet} H &\equiv \forall A \in G : \forall B \in H : A \longrightarrow B \\ G \dashrightarrow^\bullet H &\equiv \exists A \in G : \exists B \in H : A \dashrightarrow B \text{ or } A = B \end{aligned} \tag{3}$$

We can forget about the global-time models and take (3) to be the definitions of $\xrightarrow{\bullet}$ and $\dashrightarrow^\bullet$. It is easy to show that, if $\mathcal{H}$ satisfies H1 and H2 and $\longrightarrow$ and $\dashrightarrow$ satisfy A1–A5, then $\xrightarrow{\bullet}$ and $\dashrightarrow^\bullet$ also satisfy A1–A5. Therefore, if $\mathcal{H}$ is a higher-level view of S, then $\mathcal{H}, \xrightarrow{\bullet}, \dashrightarrow^\bullet$ is a system execution. If the relations $\longrightarrow$ and $\dashrightarrow$ also satisfy A#, then so do $\xrightarrow{\bullet}$ and $\dashrightarrow^\bullet$.

Let us now consider what it means for one system to implement another. If the system execution $S, \longrightarrow, \dashrightarrow$ is an implementation of a system execution $S, \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}}$, then we expect $\mathcal{H}$ to be a higher-level view of S—that is, each operation in $\mathcal{H}$ should consist of a set of operation executions of S satisfying H1 and H2. This describes the elements of $\mathcal{H}$, but not the precedence relations $\xrightarrow{\mathcal{H}}$ and $\dashrightarrow^{\mathcal{H}}$. What should those relations be?

If we consider the system execution S to be the "real" one and $\mathcal{H}$ to be a fictitious grouping of the real operation executions into abstract, higher-level ones, then the induced relations $\xrightarrow{\bullet}$ and $\dashrightarrow^\bullet$ are the "real" precedence relations on $\mathcal{H}$. These induced relations make the higher-level view $\mathcal{H}$ a system execution, so they are an obvious choice for the relations $\xrightarrow{\mathcal{H}}$ and $\dashrightarrow^{\mathcal{H}}$. However, they may not be the proper choice. Suppose that we are trying to implement an atomic register using several simpler ones, and consider a read R and write W to that register—that is, R and W are operation executions in $\mathcal{H}$ that represent a read and write to the register. Atomicity means that either $R \xrightarrow{\mathcal{H}} W$ or $W \xrightarrow{\mathcal{H}} R$. However, the two operation executions could really be concurrent. For example, there could be some operation executions A and B in the implementation of R and an operation execution C in the implementation of W with $A \longrightarrow C \longrightarrow B$, which (by (3)) implies $R \dashrightarrow^\bullet W$

and $W \dashrightarrow^{\bullet} R$. Thus, (by A2) the induced relations $\longrightarrow^{\bullet}$ and $\dashrightarrow^{\bullet}$ cannot be the desired relations $\longrightarrow^{\mathcal{H}}$ and $\dashrightarrow^{\mathcal{H}}$.

When implementing an atomic register from nonatomic ones, in addition to specifying what set of lower-level operation executions corresponds to an atomic read or write, one must also specify how to determine whether a read, which may really be concurrent with a write (according to the induced relations $\longrightarrow^{\bullet}$ and $\dashrightarrow^{\bullet}$), is considered to precede or follow that write. This must be specified in such a way that the register satisfies the condition of atomicity—namely, that each read obtains the value written by the most recent write. Subject to that requirement, there is a great deal of freedom in specifying the high-level relation $\longrightarrow^{\mathcal{H}}$.

The implementor cannot be completely free to specify the precedence relations in the high-level system any way he wishes. For example, if there is at least one write of every possible value of the register, then any system execution can be viewed as the implementation of an atomic register by choosing the $\longrightarrow^{\mathcal{H}}$ relation to be a sequential ordering of the reads and writes in which every read comes between any write of the value it read and the next write operation. This could lead to a precedence relation in which an operation is defined to precede one that really occurred several months earlier. Such a precedence relation obviously seems absurd, but why? In a real system, these reads and writes occur deep within the computer; we never actually see them happen. What is wrong with defining the precedence relation $\longrightarrow^{\mathcal{H}}$ to pretend that these operation executions happened in any order we wish? After all, we are already pretending, contrary to fact, that the operations are not concurrent.

In addition to reads and writes to registers, real systems perform externally observable operation executions such as printing on terminals. By observing these operation executions, we can infer some precedence relations among the internal reads and writes. We need some condition on $\longrightarrow^{\mathcal{H}}$ and $\dashrightarrow^{\mathcal{H}}$ to rule out precedence relations that contradict such observations.

These contradictions are avoided by requiring that the interval in which we pretend an operation execution occurs (in forming the $\longrightarrow^{\mathcal{H}}$ and $\dashrightarrow^{\mathcal{H}}$ relations) be contained within the interval in which it actually occurs. In other words, we require that a global-time model $s^{\mathcal{H}}, f^{\mathcal{H}}$ for $\mathcal{H}, \longrightarrow^{\mathcal{H}}, \dashrightarrow^{\mathcal{H}}$ satisfy

$$s^{\bullet}_A \leq s^{\mathcal{H}}_A \leq f^{\mathcal{H}}_A \leq f^{\bullet}_A \tag{4}$$

where $s^{\bullet}$ and $f^{\bullet}$ are defined by (2). To reformulate (4) directly in terms of the precedence relations, I appeal to the following result.

Proposition 4 *Let s, f be a nondegenerate global-time model for a system execution $S, \longrightarrow, \dashrightarrow$ and let $S, \xrightarrow{I}, \dashrightarrow^{I}$ be a system execution satisfying A# such that for any $A, B \in S$: $A \longrightarrow B$ implies $A \xrightarrow{I} B$. Then there exists a nondegenerate global-time model s', f' for $S, \xrightarrow{I}, \dashrightarrow^{I}$ such that, for all $A \in S$:*

$$s_A \leq s'_A < f'_A \leq f_A$$

This result implies that, if the system executions $S, \longrightarrow, \dashrightarrow$ and $\mathcal{H}, \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}}$ both satisfy A#, then the ability to choose $s^{\mathcal{H}}$ and $f^{\mathcal{H}}$ satisfying (4) is equivalent to the following condition:

H3. For any $G, H \in \mathcal{H}$: if $G \xrightarrow{\bullet} H$ then $G \xrightarrow{\mathcal{H}} H$, where $\xrightarrow{\bullet}$ is defined by (3).

This should serve to motivate the following formal definition, which does not mention global-time models.

Definition 3 *A system execution $S, \longrightarrow, \dashrightarrow$* implements *a system execution $\mathcal{H}, \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}}$ if H1–H3 are satisfied.*

To relate this definition to the preceding discussion of observable operation executions, we need the following result. Its statement relies upon the obvious fact that if $S, \longrightarrow, \dashrightarrow$ is a system execution, then $T, \longrightarrow, \dashrightarrow$ is also a system execution for any subset T of S. (The symbols $\longrightarrow$ and $\dashrightarrow$ denote both the relations on S and their restrictions to T. Also, in the proposition, the set T is identified with the set of all singleton sets $\{A\}$ for $A \in T$.)

Proposition 5 *Let $S \cup T, \longrightarrow, \dashrightarrow$ be a system execution, where S and T are disjoint; let $S, \longrightarrow, \dashrightarrow$ be an implementation of a system execution $\mathcal{H}$, $\xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}}$; and let $\xrightarrow{\bullet}$ and $\dashrightarrow^{\bullet}$ be the relations defined on $\mathcal{H} \cup T$ by (3). Then there exist precedence relations $\xrightarrow{\mathcal{H}T}$ and $\dashrightarrow^{\mathcal{H}T}$ such that:*

- *$\mathcal{H} \cup T, \xrightarrow{\mathcal{H}T}, \dashrightarrow^{\mathcal{H}T}$ is a system execution that is implemented by $S \cup T, \longrightarrow, \dashrightarrow$.*
- *The restrictions of $\xrightarrow{\mathcal{H}T}$ and $\dashrightarrow^{\mathcal{H}T}$ to $\mathcal{H}$ equal $\xrightarrow{H}$ and $\dashrightarrow^{H}$, respectively.*
- *The restrictions of $\xrightarrow{\mathcal{H}T}$ and $\dashrightarrow^{\mathcal{H}T}$ to T are extensions of the relations $\xrightarrow{\bullet}$ and $\dashrightarrow^{\bullet}$, respectively.*

To apply this proposition to our discussion of implementations, let S, $\longrightarrow$,- - → be an execution of a lower-level system of register reads and writes implementing a higher-level system execution H,$\xrightarrow{H}$,- $\overset{H}{-}$ → of reads and writes. Let T be the set of all other operation executions in the system, including the observable ones. Proposition 5 means that, while the precedence relations $\xrightarrow{H}$ and - $\overset{H}{-}$ → may imply new precedence relations on the operation executions in T, these relations ($\xrightarrow{HT}$ and - $\overset{HT}{-}$ →) are consistent with the "real" precedence relations $\xrightarrow{*}$ and - $\overset{*}{-}$ → on T.

Note that, when there are global-time models for all the system executions, the * relations are the same as the original precedence relations on the set T, and Proposition 4 implies that the HT relations can be chosen also to be the same as the original precedence relations on T. However, in general, the relation $\xrightarrow{H}$ may contain orderings that imply additional orderings on the elements of T beyond those contained in $\xrightarrow{*}$. As a simple example, let $A, B \in S$, let $S,T \in T$, let $S \longrightarrow A$, $B \longrightarrow T$ be the only precedence relations among these elements, and let $H = S$. If $A \xrightarrow{H} B$, then A1 implies $S \xrightarrow{HT} T$ even though $S \overset{*}{\not\longrightarrow} T$.

When implementing a register, I will ignore any operation executions not involved in the implementation, and consider the system execution comprising only the reads and writes that implement the register. Proposition 5 shows that the implementation cannot lead to any anomalous precedence relations among the operation executions that are being ignored.

An implementation S,$\longrightarrow$,- - → of H,$\xrightarrow{H}$,- $\overset{H}{-}$ → is said to be *trivial* if every element of H is a singleton set. In other words, a trivial implementation is one in which each higher-level operation execution is implemented by a single lower-level one. In a trivial implementation, the sets S and H are (essentially) the same; the two system executions differ only in their precedence relations.

Proposition 3 implies that any system execution trivially implements one that satisfies A#, which, by Proposition 2, has a global-time model. Implementation is transitive—if S,$\longrightarrow$,- - → implements S',$\xrightarrow{\prime}$,- $\overset{\prime}{-}$ → which in turn implements H,$\xrightarrow{H}$,- $\overset{H}{-}$ →, then S,$\longrightarrow$,- - → implements H,$\xrightarrow{H}$,- $\overset{H}{-}$ →. When implementing a higher-level system, we can therefore assume the lower-level system execution has a global-time model. However, there is no reason to do so; a rigorous correctness proof using Axioms A1–A5 will be at least as simple as one based upon starting and finishing times, and will be more reliable than an intuitive one based upon pictures of intervals.

3.3 Register Axioms

The foregoing discussion applies to any system execution. I now consider system executions containing reads and writes to registers. In addition to A1–A5, some axioms special to these kinds of operation executions are needed, including axioms that provide the formal definitions of safe, regular, and atomic registers.

Axioms A1–A5 do not require that there be any precedence relations among operation executions. However, some precedence relation between a read and a write to the same register must be assumed. (Communication requires a causal connection between reads and writes.) The following axiom is assumed; the reader is referred to [5] (where it is labeled C3) for its justification. Note that it is implied by A#.

B1. For any read R and write W to the same register, $R \dashrightarrow W$ or $W \dashrightarrow R$ (or both).

Each register is assumed to have a finite set of possible values—for example, a boolean-valued register has the possible values *true* and *false*. I assume that any read, whether or not it overlaps a write, obtains one of these values.

B2. A read of a register obtains one of the values that may be written in the register.

Thus, a read of a Boolean register cannot obtain a nonsense value like "*trlse*". This axiom does not assume that the value obtained by a read was ever actually written in the register.

I assume that a register v is written by only a single writer and that each write precedes the next. Let $V^{[1]}, V^{[2]}, \ldots$ denote the sequence of write operations to the register v, where

$$V^{[1]} \longrightarrow V^{[2]} \longrightarrow \ldots$$

and let $v^{[i]}$ denote the value written by $V^{[i]}$. (There may be a finite or infinite number of write operations $V^{[i]}$.)

A register v is assumed to have some initial value $v^{[0]}$. It is convenient to assume that this value is written by a write $V^{[0]}$ that precedes ($\longrightarrow$) all other reads and writes of v. Eliminating this assumption changes none of the results, but it complicates the reasoning because a read that precedes all writes has to be treated as a separate case.

Let R be a read of register v, and let

$$I_R \stackrel{\text{def}}{=} \{V^{[k]} : R \not\rightarrow V^{[k]}\}$$
$$J_R \stackrel{\text{def}}{=} \{V^{[k]} : V^{[k]} \dashrightarrow R\}$$

From A2 and the assumption that $V^{[0]}$ precedes all reads, it follows that $V^{[0]}$ is in both I_R and J_R; and from A2 and A5 it follows that I_R and J_R are finite. The writes in J_R are the ones that could affect R. For the sake of the following intuitive discussion, suppose that A# holds, so I_R is the set of writes that precede ($\longrightarrow$) R. (The reader interested in extending his intuition to the general case should substitute "effectively precedes" for "precedes"—a concept defined in [5].) The difference $J_R - I_R$ of these two sets is the set of writes concurrent with R. The read R can observe "traces" of the values written by writes in $J_R - I_R$, and by the last write in I_R. All traces of earlier writes are assumed to vanish with the completion of the last write in I_R, and no write later than the last one in J_R can influence R in any way.

I will say that R *sees* $v^{[i,j]}$ if it can observe traces of the writes $V^{[i]}$ through $V^{[j]}$. The formal definition is as follows:

Definition 4 *A read R of register v is said to* see $v^{[i,j]}$ *where:*

$$i \stackrel{\text{def}}{=} \max\{k : R \not\rightarrow V^{[k]}\}$$
$$j \stackrel{\text{def}}{=} \max\{k : V^{[k]} \dashrightarrow R\}$$

This definition makes sense because i and j are defined to be the maxima of finite, nonempty sets—A5 and A2 imply that they are finite, and they both contain zero. Also observe that B1 implies that $i \leq j$.

I can now give the formal definitions of safe, regular, and live registers. A safe register is one that obtains the correct value if it is not concurrent with any write. This is the case if it observes traces of only a single write.

B3. (*safe*) A read that sees $v^{[i,i]}$ obtains the value $v^{[i]}$.

A regular register is one that obtains a value that it "could have" seen.

B4. (*regular*) A read that sees $v^{[i,j]}$ obtains a value $v^{[k]}$ for some k with $i \leq k \leq j$

An atomic register satisfies the additional requirement that a read is never concurrent with any write.

B5. (*atomic*) If a read sees $v^{[i,j]}$ then $i = j$.

A safe register satisfies B1–B3, a regular register satisfies B1–B4 (note that B4 implies B3), and an atomic register satisfies B1–B5.

The following two propositions state some useful properties that are simple consequences of Definition 4. I introduce the notation of letting $v^{[i,j]}$ stand for a read that sees the value $v^{[i,j]}$. Thus, part (a) is an abbreviation for: "If R is a read that sees $v^{[i,j]}$ and $R \longrightarrow V^{[k]}$ then ...." (Recall that $V^{[k]}$ is the k^{th} write of v.)

Proposition 6 (a) *If* $v^{[i,j]} \longrightarrow V^{[k]}$ *then* $j < k$.

(b) *If* $V^{[k]} \longrightarrow v^{[i,j]}$ *then* $k \leq i$.

(c) *If* $v^{[i,j]} \longrightarrow v^{[i',j']}$ *then* $j \leq i' + 1$.

Proposition 7 *If R is a read that sees* $v^{[i,j]}$, *then*

(a) $k \leq j$ *if and only if* $V^{[k]} \dashrightarrow R$.

(b) $i \leq k$ *if and only if* $R \dashrightarrow V^{[k+1]}$.

In a global-time view, atomicity is usually defined to mean that all operations are instantaneous. In B5, it is defined by the requirement that a write does not overlap a read. However, two reads may overlap, and a write could overlap some operation execution that is not a read or write of the register. It is easy to see that, given a global-time model for a system execution satisfying B5, without violating conditions B1–B5, we can shrink the intervals occupied by reads and writes so that they overlap no other operations. Thus, the original system execution implements one in which reads and writes of the atomic register are instantaneous.

For a nonatomic register, reads and writes cannot be made instantaneous. However, the reads can be made instantaneous.

Proposition 8 *Any system execution* $S, \longrightarrow, \dashrightarrow$ *having a safe or regular register v trivially implements a system execution* $S, \xrightarrow{!}, \overset{!}{\dashrightarrow}$ *in which v is also safe or regular, such that* $S, \xrightarrow{!}, \overset{!}{\dashrightarrow}$ *has a global-time model in which every read of v is instantaneous.*

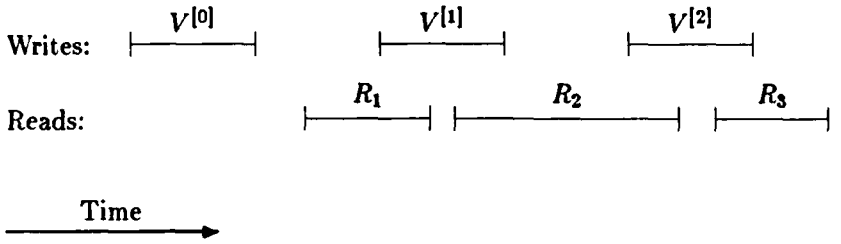


Figure 2: An interesting collection of reads and writes.

I have observed that a regular register is not necessarily atomic because two successive reads that overlap the same write could return the new then the old value. The following result shows that this is the only way a regular register can fail to be atomic.

Proposition 9 *Let $S, \longrightarrow, \dashrightarrow$ be a system execution containing reads and writes to a regular register v, and let ϕ be an integer-valued function on the set of reads such that:*

1. *If R sees $v^{[i,j]}$, then $i \leq \phi(R) \leq j$.*
2. *A read R returns the value $v^{[\phi(R)]}$.*
3. *If $R \longrightarrow R'$ then $\phi(R) \leq \phi(R')$.*

Then $S, \longrightarrow, \dashrightarrow$ trivially implements a system execution in which v is an atomic register.

A function ϕ satisfying the first two properties exists if and only if v is regular. One might be tempted to replace these three properties with the requirement that v be regular and that the following hold:

3′ If $v^{[i,j]} \longrightarrow v^{[i',j']}$, then there exist k and k' with $i \leq k \leq j$ and $i' \leq k' \leq j'$ such that $v^{[i,j]}$ returns the value $v^{[k]}$ and $v^{[i',j']}$ returns the value $v^{[k']}$.

However, this does not imply atomicity. As a counterexample, let $v^{[0]} = v^{[2]} = 0$ and $v^{[1]} = 1$, let R_1, R_2, R_3 be the three reads shown in Figure 2, and suppose that R_1 and R_3 return the value 1 while R_2 returns the value 0. The reader can show that this register is regular, but no such ϕ can be constructed; there is no way to interpret these reads and writes as belonging to an atomic register while maintaining the given orderings among the writes and among the reads.

If two reads cannot overlap the same write, then $v^{[i,j]} \longrightarrow v^{[i',j']}$ implies $j \leq i'$. This implies that any ϕ satisfying conditions 1 and 2 of Proposition 9 also satisfies condition 3. But such a ϕ exists if v is regular, so any regular register trivially implements an atomic one if two reads cannot overlap a single write.

3.4 Systems

I have defined a system execution, but not a system. Formally, a system is just a set of system executions—a set that represents all possible executions of the system.

Definition 5 *A* system *is a set of system executions. The system* **S** *is said to contain a register v satisfying one or more of the properties B1-B5 if every system execution in* **S** *contains a sequence $V^{[1]} \longrightarrow \cdots$ of writes with associated values $v^{[1]}, \ldots$ and a set of reads satisfying the corresponding properties.*

The usual method of describing a system is with a program written in some programming language. Each execution of such a program describes a system execution, and the program represents the system consisting of the set of all such executions. The only operation executions that concern us are reads and writes of a register; "calculation" steps can be ignored. For example, execution of the statement $x := y \vee z$ includes three operation executions: a read of y, a read of z, and a write of x. It does not matter whether or not the computation of the $\vee$ is considered to be a separate operation execution. What is significant is that each of the two reads precedes ($\longrightarrow$) the write; no precedence relation is assumed between the two reads.

A formal semantics for a programming language can be given by defining, for each syntactically correct program, the set of all possible executions. This is done by recursively defining a succession of lower and lower higher-level views, in which each operation execution represents a single execution of a syntactic program unit.[2] At the highest-level view, a system execution consists of a single operation execution that represents an execution of the entire program. A view in which an execution of the statement $S;T$ is a single operation execution is refined into one in which an execution consists

[2]For nonterminating programs, the formalism must be extended to allow a nonterminating higher-level operation execution that consists of an infinite set of lower-level operation executions.

of an execution of S followed by ($\longrightarrow$) an execution of T.[3] While this kind of formal semantics may be useful in studying subtle programming language issues, it is unnecessary for the simple language constructs used in the algorithms of this paper, so I will just employ these ideas informally.

Having defined what a system is, I should define what it means for one system to implement another. The definition is, of course, in terms of the definition of what it means for one system execution to implement another.

Definition 6 *The system* **S** implements *a system* **H** *if there is a mapping* $\iota : \mathbf{S} \mapsto \mathbf{H}$ *such that, for every system execution* $S, \longrightarrow, \dashrightarrow$ *in* **S**, $S, \longrightarrow, \dashrightarrow$ *implements* $\iota(S, \longrightarrow, \dashrightarrow)$.

Note that for **S** to implement **H**, every execution of **S** must correspond to some execution of **H**. The converse is not required; I do not insist that every possible execution of **H** have a corresponding implementation. A higher-level description **H** of a system can be viewed as a specification of its implementation—a specification that describes all allowed behaviors, but does not require any particular behavior.

This definition raises the question of how we can specify that the system must actually do anything. The specification of a banking system must allow a possible system execution in which no customers happen to use an automatic teller machine on a particular afternoon, and it must include the possibility that a customer will enter an invalid request. How can we rule out an implementation in which the machine simply ignores all customer requests during an afternoon, or interprets any request as an invalid one?

The answer lies in the concept of an *interface specification*, discussed in [8]. The specification must explicitly describe how certain interface operations are to be implemented; their implementation is not left to the implementor. The interface specification for the bank includes a description of what sequences of keystrokes at the teller machine constitute valid requests, and the set of system executions only includes ones in which every valid request is serviced. What it means for someone to use the machine is part of the interface specification, so the possibility of no one using the machine on some afternoon does not allow the implementation to ignore someone who does use it.

Since this paper considers only the internal operations that effect communication between processes within the system, not the interface operations that effect communication between the system and its environment, I

[3] In the general case, we must also allow the possibility that an execution of $S;T$ consists of a nonterminating execution of S.

will ignore interface specifications. The interested reader is referred to [8] for a discussion of this subject.

4 Correctness Proofs for the Constructions

4.1 Proof of Constructions 1, 2, and 3

These constructions are all simple, and the correctness proofs are essentially trivial. Formal proofs add no further insight into the constructions, but they do illustrate how the formalism developed in the preceding section is applied to actual algorithms. I therefore indicate all the formal details in the proof of Construction 1. The formal proofs for the other two constructions are just briefly sketched.

Recall that, in Construction 1, the m-reader register v is implemented by the m single-reader registers v_i. Formally, this construction defines a system, which I denote by **S**, that is the set of all system executions consisting of reads and writes of the v_i such that the only operations to these registers are the ones indicated by the readers' and writer's programs. Thus, **S** consists of all system executions $S, \longrightarrow, \dashrightarrow$ such that:

- S consists of reads and writes of the registers v_i.
- Each v_i is written by the same writer and is read only by the i^{th} reader.
- For any i and j: if the write $V_i^{[k]}$ occurs, then the write $V_j^{[k]}$ also ocurs, and $v_i^{[k-1]} \longrightarrow v_j^{[k]}$.

The third condition expresses the formal semantics of the writer's algorithm, asserting that a write of v is done by writing all the v_i, and that a write of v is completed before the next one is begun.

To say that the v_i are safe or regular means that the system **S** is further restricted to contain only system executions that satisfy B1–B3 or B1–B4, when each v_i is substituted for v in those conditions.

To show that this construction implements a register v, Definition 6 states that we must construct a mapping ι from **S** to the system **H**, which consists of the set of all system executions formed by reads and writes to an m-reader register v. To say that v is safe or regular means that **H** contains only system executions satisfying B1–B3 or B1–B4.

In giving the readers' and writer's algorithms, the construction implies that, for each system execution $S, \longrightarrow, \dashrightarrow$ of **S**, the set $\iota(S)$ of operation

executions of $\iota(S, \longrightarrow, \text{- - }\rightarrow)$ is the higher-level view of $S, \longrightarrow, \text{- - }\rightarrow$ consisting of all writes $V^{[k]}$ of the form $\{V_1^{[k]}, \ldots, V_m^{[k]}\}$, for $V_i^{[k]} \in S$, and all reads of the form $\{R_i\}$, where $R_i \in S$ is a read of v_i. (The write $V^{[k]}$ exists in $\iota(S)$ if and only if some, and hence all, $V_i^{[k]}$ exists.) Conditions H1 and H2 are obviously satisfied, so this is indeed a higher-level view. To complete the mapping ι, we must define the precedence relations $\xrightarrow{N}$ and $\text{- -}\overset{N}{\rightarrow}$ so that $\iota(S, \longrightarrow, \text{- - }\rightarrow)$ is defined to be $\iota(S), \xrightarrow{N}, \text{- -}\overset{N}{\rightarrow}$. Proving the correctness of the construction means showing that:

1. $\iota(S), \xrightarrow{N}, \text{- -}\overset{N}{\rightarrow}$ is a system execution—that is, it satisfies A1–A5.

2. $S, \longrightarrow, \text{- - }\rightarrow$ implements $\iota(S), \xrightarrow{N}, \text{- -}\overset{N}{\rightarrow}$—that is, H1–H3 are satisfied.

3. $\iota(S), \xrightarrow{N}, \text{- -}\overset{N}{\rightarrow}$ is in **H**—that is, B1–B3 or B1–B4 are satisfied.

The precedence relations on $\iota(S)$ are defined to be the "real" ones, with $G \xrightarrow{N} H$ if and only if G really precedes H. Formally, this means that we let $\xrightarrow{N}$ and $\text{- -}\overset{N}{\rightarrow}$ be the induced relations $\xrightarrow{*}$ and $\text{- -}\overset{*}{\rightarrow}$, defined by (3). Recall from Section 3.2 that the induced precedence relations make any higher-level view a system execution, so 1 is satisfied. I have already observed that H1 and H2, which are independent of the choice of precedence relations, are satisfied, and H3 is trivially satisfied by the induced precedence relations, so 2 holds. Therefore, we need only show that, if B1–B3 or B1–B4 are satisfied for reads and writes of each of the registers v_i in $S, \longrightarrow, \text{- - }\rightarrow$, then they are also satisfied by the register v of $\iota(S), \xrightarrow{N}, \text{- -}\overset{N}{\rightarrow}$.

Property B1 for $\iota(S), \xrightarrow{*}, \text{- -}\overset{*}{\rightarrow}$ follows easily from, (3) and property B1 for $S, \longrightarrow, \text{- - }\rightarrow$. Property B2 is immediate. The informal proof of B3 is as follows: if a read of v by process i does not overlap a write (in $\iota(S)$), then the read of v_i does not overlap any write of v_i, so it obtains the correct value. A formal proof is based upon:

X. If a read R_i in $S, \longrightarrow, \text{- - }\rightarrow$ sees $v_i^{[k,l]}$, then the corresponding read $\{R_i\}$ in $\iota(S), \xrightarrow{*}, \text{- -}\overset{*}{\rightarrow}$ sees $v^{[k',l']}$, where $k' \leq k \leq l \leq l'$.

The proof of X is a straightforward application of (3) and Defintion 4. Property X easily implies that, if B3 or B4 holds for $S, \longrightarrow, \text{- - }\rightarrow$, then it holds for $\iota(S), \xrightarrow{*}, \text{- -}\overset{*}{\rightarrow}$. This completes the formal proof of Construction 1.

The formal proof of Construction 2 is quite similar. Again, the induced precedence relations are used to turn a higher-level view into a system execution. The proof of Construction 3 is a bit trickier because a write operation

to v^* that does not change its value consists only of the read operation to the internal variable x. This means that the induced precedence relations do not necessarily satisfy B1; they must be extended to make B1 hold. This can be done by applying Proposition 3, though a more "economical" extension can also be constructed.

4.2 Proof of Construction 4

The higher-level system execution of reads and writes to v is defined to have the induced precedence relations $\xrightarrow{*}$ and $\dashrightarrow^{*}$. As in the above proofs, verifying that this defines an implementation and that B1 holds is trivial. The only problems are proving B2—namely, showing that the reader must find some v_i equal to one—and proving B4 (which implies B3).

I first prove the following property:

Y. If a read returns the value μ, then there is some k such that $v^{[k]} = \mu$, and the read sees $v^{[l,r]}$ with $l \leq k \leq r$.

If B2 holds, then property Y implies B4.

Reasoning about the construction is complicated by the fact that a write of v does not write all the v_j, so the write of v_j that occurs during the k^{th} write of v is not necessarily the k^{th} write of v_j. To overcome this difficulty, I introduce new names for the write operations to the v_j. If v_j is written during the execution of $V^{[k]}$, then I let $W_j^{[k]}$ denote that write of v_j; otherwise, $W_j^{[k]}$ is undefined. Thus, every write $V_j^{[l]}$ of v_j is also named $W_j^{[l']}$ for some $l' \geq l$. I will say that a read of v_j sees $w_j^{[l',r']}$ if it sees $v^{[l,r]}$ and the writes $W_j^{[l']}$ and $W_j^{[r']}$ are the same writes as $V_j[l]$ and $V_j[r]$, respectively. Note that, because the writer's algorithm writes from "right to left", if $W_i^{[k]}$ exists, then so do all the $W_j^{[k]}$ with $j < i$. In particular, $W_1^{[k]}$ exists for all k.

Let R be a read that returns the value μ, and let μ be the i^{th} value, so R consists of the sequence of reads $R_1 \longrightarrow \cdots \longrightarrow R_i$, where each R_j is a read of v_j. All the R_j return the value 0 except R_i, which returns the value 1. Let R see $v^{[l,r]}$ and let each R_j see $w_j^{[l(j),r(j)]}$. By regularity of v_j, there is some $k(j)$ with $l(j) \leq k(j) \leq r(j)$ such that $W_i^{[k(i)]}$ writes a 1 and $W_j^{[k(j)]}$ writes a 0 for $1 \leq j < i$. Thus, $v^{[k(i)]}$ is the value read by R, so it suffices to show that $l \leq k(i) \leq r$.

Definition 4 implies $W_i^{[r(i)]} \dashrightarrow R_i$, which by (3) implies $V^{[r(i)]} \dashrightarrow^{*} R$, which implies $r(i) \leq r$. Hence, $k(i) \leq r$.

For any p with $p \leq l$, Definition 4 implies that $R \overset{\bullet}{\not\dashrightarrow} V^{[p]}$, which implies that $R_1 \not\dashrightarrow W_1^{[p]}$, which in turn implies that $p \leq l(1)$. Hence, $l \leq l(1)$.[4] Since $l(j) \leq k(j)$, it suffices to prove that $k(j) \leq l(j+1)$ for $1 \leq j < i$.

Since $k(j) \leq r(j)$, Definition 4 implies that $W_j^{[k(j)]} \dashrightarrow R_j$. Because $W_j^{[k(j)]}$ writes a zero, $W_{j+1}^{[k(j)]}$ exists, and we have

$$W_{j+1}^{[k(j)]} \longrightarrow W_j^{[k(j)]} \dashrightarrow R_j \longrightarrow R_{j+1}$$

where the two $\longrightarrow$ relations are implied by the order in which writing and reading of the individual v_j are performed. By A4, this implies that $W_{j+1}^{[k(j)]} \longrightarrow R_{j+1}$, which, by A2, implies $R_{j+1} \not\dashrightarrow W_{j+1}^{[k(j)]}$. By Definition 4, this implies that $k(j) \leq l(j+1)$, completing the proof of property Y.

To complete the proof of the construction, I must only prove that every read does return a value. Let R and the values $l(j)$, $k(j)$, and $r(j)$ be as above, except let $i = n$ and drop the assumption that R_i obtains the value 1. To prove B2, I must prove that R_n does obtain the value 1.

The same argument used above shows that,if R_j obtains a zero, then that zero was written by some write $W_j^{[k(j)]}$, which implies that $W_{j+1}^{[k(j)]}$ exists and $k(j) \leq l(j+1)$. Since R_n obtains the value written by $W_n^{[k(n)]}$, it must obtain a 1 unless $k(n) = 0$ and the initial value is not the n^{th} one. Suppose the initial value $v^{[0]}$ is the p^{th} value, encoded with $v_p = 1$, $p < n$. Since R_p obtains the value 0, we must have $k(p) > 0$, which implies that $k(n) > 0$, so R_n obtains the value 1. This completes the proof of the construction.

4.3 Proof of Construction 5

This construction defines a set $\mathcal{H}$, consisting of reads and writes of v^*, that is a higher-level view of a system execution $S, \longrightarrow, \dashrightarrow$ whose operation executions are reads and writes of the two shared registers v, cw and cr. As usual, $\overset{\bullet}{\longrightarrow}$ and $da*$ denote the induced precedence relations on S that are defined by (3).

Let u denote the shared register v, cw of the algorithm. In this construction, the write $V^{*[k]}$ of v^*, for $k > 0$, is implemented by the sequence $R \longrightarrow U^{[2k-1]} \longrightarrow U^{[2k]}$, where R is a read of cr and $U^{[i]}$ is the i^{th} write of u. The initial write $V^{*[0]}$ of v^* is just the initial write $U^{[0]}$ of u.

[4] Note that the same argument does not prove that $l \leq l(i)$ because $W_i^{[p]}$ does not necessarily exist.

Since there is only one reader, the reads of v^* are totally ordered by $\xrightarrow{*}$. The i^{th} read S_i of v^* consists of the sequence $R_i \longrightarrow CR^{[i]}$ where R_i is the i^{th} read of u and $CR^{[i]}$ is the i^{th} write of cr. For notational convenience, I assume an imaginary read R_0 of u that returns the value $u^{[0]}$, and I define S_0 to be the sequence of operations $R_0 \longrightarrow CR^{[0]}$. The operation S_0 is taken to be the one that sets the initial values of x' and cr'.

The proof of correctness is based upon Proposition 9. Letting $\phi(i)$ denote $\phi(S_i)$, to apply that proposition, it suffices to choose the $\phi(i)$ such that the following three properties hold:

- S_i returns the value $v^{*[\phi(i)]}$.
- If S_i sees $v^{*[l,r]}$ then $l \leq \phi(i) \leq r$.
- If $j < i$ then $\phi(j) \leq \phi(i)$.

I start by defining a function ψ such that R_i returns the value $u^{[\psi(i)]}$ and, if R_i sees $u^{[l,r]}$ then $l \leq \psi(i) \leq r$. Since u is regular, such a ψ exists. Proposition 6 implies:

Z1. If $j < i$ then $\psi(j) \leq \psi(i) - 1$.

By Proposition 7, $U^{[\psi(i)]} \dashrightarrow R_i \dashrightarrow U^{[\psi(i)+1]}$. Suppose $\psi(i) = 2k$. Since $U^{[2k]}$ is part of $V^{*[k]}$, $U^{[2k+1]}$ is part of $V^{*[k+1]}$, and R_i is part of S_i, this implies $V^{*[k]} \dashrightarrow S_i \dashrightarrow V^{*[k+1]}$. Hence, property 2 is satisfied if $\phi(i) = k$. Next, suppose that $\psi(i) = 2k - 1$, where $k > 0$. Since $V^{[2k-1]}$ is part of $V^{*[k]}$, we have $V^{*[k]} \dashrightarrow S_i \dashrightarrow V^{*[k]} \xrightarrow{*} V^{*[k+1]}$, so property 2 is satisfied if $\phi(i) = k - 1$. But we also have $V^{*[k-1]} \xrightarrow{*} V^{*[k]} \dashrightarrow R_i$, so property 2 is also satisfied if $\phi(i) = k - 1$. To summarize, property 2 is satisfied by i if the following holds:

Z2. (a) If $\psi(i) = 2k$ then $\phi(i) = k$.
(b) If $\psi(i) = 2k - 1$ then $\phi(i) = k$ or $\phi(i) = k - 1$.

The second statement in the algorithm of Figure 1 consists of nested **if** statements, so executing it executes exactly one innermost **then** or **else** clause. I will use a sequence of **t** (for **then**) and **e** (for **else**) characters to denote such an innermost clause; for example, **tee** denotes the second innermost **else** clause, which is executed if $x_1 \neq x_2$ and $x_1' = x_2' = x_2$.

Let a *ttt-read* be one that executes the **ttt** clause of the reader's algorithm, and let a *nice* read be one that is not a ttt-read. The initial read S_0 is defined to be nice. For any $i > 0$, let $\pi(i)$ denote the largest integer such

that $\pi(i) < i$ and $S_{\pi(i)}$ is nice. In other words, $S_{\pi(i)}$ is the last nice read before S_i. A ttt-read does not change the value of rtn, x', or cr'. Therefore, when the execution of S_i begins, rtn has the value returned by $S_{\pi(i)}$ and x', cr' has the value $u^{[\psi(\pi(i))]}$ read by $R_{\pi(i)}$.

I first define $\phi(i)$ inductively for all nice reads, starting with $\phi(0) = 0$. The definition will be made so that Z2 holds for all i. Let i be a nice read, $i > 0$, and assume that properties 1–3 and Z2 hold with $\pi(i)$ substituted for i. In the following discussion, I will refer to the values of variables immediately after the execution of the first statement in the reader's algorithm during the operation execution S_i. Thus, x, cr is the value $u^{[\psi(i)]}$ read by R_i, rtn is the value $v^{*[\phi(\pi(i))]}$ returned by $S_{\phi(i)}$, and x', cr' is the value $u^{[\psi(\pi(i))]}$ read by $R_{\pi(i)}$.

Consider first the case $\psi(i) = 2k - 1$. In this case, $x_1 = v^{*[k-1]}$ and $x_2 = v^{*[k]}$. If $x_1 \neq x_2$, then properties 1 and Z2 are satisfied only by defining $\phi(i)$ to equal $k - 1$ if S_i returns the value x_1 and to equal k if S_i returns the value x_2. In other words, $\phi(i)$ equals k if S_i executes the **tet** clause and equals $k - 1$ otherwise. Since Z2 is satisfied, property 2 holds.

To prove property 3 for i, it suffices to prove that $\phi(\pi(i)) \leq \phi(i)$, since property 3 is assumed to hold for $\pi(i)$. Property Z1 implies that $\psi(\phi(i)) \leq 2k$, so Z2 implies that $\phi(\pi(i))$ can be greater than $\phi(i)$ only in two cases: (i) $\psi(\pi(i)) = 2k$ and $\phi(i) = k - 1$, or (ii) $\psi(\pi(i)) = 2k - 1$, $\phi(\pi(i)) = k$, and $\phi(i) = k - 1$. But $\psi(\pi(i)) = 2k$ implies that $x'_1 = x'_2 = x_2$, so S_i executes the **tet** clause and $\phi(i) = k$. Hence, case (i) is impossible. If $\psi(\pi(i)) = 2k - 1$ and $\phi(i) = k$, then $x' = x$ and $S_{\phi(i)}$ executes the **tet** clause, so $rtn' = x'_2$. Hence, S_i must also execute the **tet** clause, so $\phi(i) = k$, showing that case (ii) is impossible. This completes the case $\psi(i) = 2k - 1$ and $x_1 \neq x_2$.

If $\psi(i) = 2k - 1$ and $x_1 = x_2$, then I define $\phi(i)$ to be the maximum of $k - 1$ and $\phi(\pi(i))$. Z1 and Z2 (for $\pi(i)$) imply that $\phi(\pi(i)) \leq k$, so this defines $\phi(i)$ to equal either $k - 1$ or k. At this point, I note the following property for later use:

Z3. If $\psi(i) = 2k - 1$, $x_1 = x_2$, and $\phi(i) = k$, then there is a nice read R_j with $j < i$ such that $\psi(j) = 2k$.

The proof of Z3 is by induction on i. The hypothesis Z1, and Z2 imply that either $\psi(\pi(i)) = 2k$, in which case we can let $j = \pi(i)$, or else $\psi(\pi(i)) = 2k - 1$ and $\phi(\pi(i)) = k$, in which case we apply Z3 with $\pi(i)$ substituted for i.

Returning to the definition of $\phi(i)$, in the case under consideration ($\psi(i) = 2k - 1$ and $x_1 = x_2$), properties 1, 2, and Z2 are satisfied because

$\phi(i)$ equals either $k-1$ or k. Moreover, we obviously have $\phi(\pi(i)) < \phi(i)$, so property 3 is also satisfied. This completes the case $\psi(i) = 2k-1$ and $x_1 \neq x_2$.

Finally, I consider the case $\psi(i) = 2k$, where $\phi(i)$ must be defined to equal k to satisfy Z2. In this case, $x_1 = x_2 = v^{*[k]}$ and S_i executes the **tte** clause, returning the value x_1. (Since S_i is assumed to be nice, it does not execute the **ttt** clause.) Hence, property 1 is satisfied. Since Z2 holds, property 2 is satisfied. To prove property 3 for i, it suffices to show that $\phi(\pi(i)) \leq \phi(i)$, since the property holds for $\pi(i)$. By Z1, $\psi(\pi(i)) \leq 2k+1$, so $\phi(\pi(i))$ can be greater than $\phi(i)$ only if $\psi(\pi(i)) = 2k+1$ and $\phi(\pi(i)) = k+1$. There are two possibilities to consider: (i) $x'_1 \neq x'_2$ and (ii) $x'_1 = x'_2$. In case (i), $\phi(\pi(i))$ can equal $k+1$ only if $S_{\pi(i)}$ executes the **tet** clause, which implies that $x'_1 \neq x'_2$ and $rtn = x'_2$; but this is impossible since S_i executes the **tte** clause. In case (ii), Z3 implies that, if $\phi(\pi(i)) = k+1$, then there exists $j < \pi(i)$ with $\psi(j) = 2k+2$. But Z1 implies that this is impossible, since $j < i$ and $\psi(i) = 2k$. Hence, property 3 holds. This completes the construction of $\phi(i)$ for all nice reads S_i.

To complete the definition of ϕ, if S_i is a ttt-read, I define $\phi(i)$ to equal $\phi(\pi(i))$. Since S_i returns the same value as $S_{\pi(i)}$, property 1 is satisfied. Property 3 obviously holds, since it holds for nice reads and ϕ assigns to every ttt-read the same value as it assigns the most recent nice read. The only thing left to prove is that property 2 holds for a ttt-read S_i. This is perhaps the most subtle proof of the entire paper. It involves proving the remark made earlier, that, if a sequence of reads obtains the values (μ,μ), (ν,μ), and (ν,ν), all of the same color, then the last read overlaps the write of (ν,μ).

Let S_i be a ttt-read, and let $(\mu,\mu),c$ be the value $u^{[\psi(i)]}$ read by R_i. Since S_i executes the **ttt** clause, x', cr', which is the value $u^{[\psi(\pi(i))]}$ read by $R_{\pi(i)}$, must equal $(\nu,\mu),c$ for some $\nu \neq \mu$, so $\psi(\pi(i))$ is odd. Let $\psi(\pi(i)) = 2k-1$. Since S_i executes the **ttt** clause, $S_{\pi(i)}$ must return μ, so it must execute the **tet** clause. This implies that $\phi(\pi(i)) = k$, so $\phi(i) = k$, and that the value of cw read by the operation execution $S_{\pi(i)-1}$ must also equal c, so $CR^{[\pi(i)-1]}$ writes the value c. The following operation executions must therefore be performed in sequence by the reader (each one $\longrightarrow$'s the next, but the reader may perform other, intervening operation executions):

- $CR^{[\pi(i)-1]}$: writes $cr[\pi(i)-1] = c$
- $R_{\pi(i)}$: reads $u^{[2k-1]} = (\nu,\mu),c$

- R_i: reads $u^{[\psi(i)]} = (\nu, \nu), c$
- $CR^{[i]}$: writes $cr^{[i]} = c$

Moreover, the reads between $S_{\pi(i)}$ and S_i also write the value c in cr. Therefore, $cr^{[j]} = c$ for all j with $\pi(i) - 1 \leq j \leq i$. Note also that $\phi(i) = \phi(\pi(i)) = k - 1$.

It follows from Z1 that $\psi(i) \geq 2k-2$. If $\psi(i) = 2k-2$, then Proposition 7 implies that $R_i \dashrightarrow U^{[2k-1]}$. However, that proposition also implies that $U^{[2k-1]} \dashrightarrow R_{\pi(i)}$. Since $U^{[2k-2]} \longrightarrow U^{[2k-1]}$ and $R_{\pi(i)} \longrightarrow R_i$, we see that $U^{[2k-2]} \longrightarrow R_i \dashrightarrow U^{[2k-1]}$. This implies $V^{*[k-1]} \stackrel{*}{\dashrightarrow} S_i \stackrel{*}{\dashrightarrow} V^{*[k]}$. Since $\phi(i) = k - 1$, property 2 follows from Proposition 7.

I have shown that $\psi(i) \geq 2k - 2$ and property 2 holds if $\psi(i) = 2k - 2$. To finish the proof, I now show that $\psi(i) = 2k - 2$ by assuming $\psi(i) > 2k - 2$ and obtaining a contradiction. Since $u^{[2k-1]}$ equals $(\nu, \mu), c$ and $U^{[2k]}$ equals (μ, μ), neither of which equals $u^{[\psi(i)]}$ (because $\mu \neq \nu$), we must have $\psi(i) > 2k$. Let $cr^{[l,r]}$ denote the read of cr in the write of v^* of which $U^{[\psi(i)]}$ is a part. Since $U^{[\psi(i)]}$ sets cw to c, the read $cr^{[l,r]}$ must obtain the value $\neg c$. The writer must therefore perform the following sequence of operation executions, where each $\longrightarrow$'s the next. (There may be other, intervening operation executions.)

- $U^{[2k]}$: writes $u^{[2k]} = (\mu, \mu), c$
- $cr^{[l,r]}$: reads the value $\neg c$
- $U^{[\psi(i)]}$: writes $u^{[\psi(i)]} = (\nu, \nu), c$

By Proposition 7 (and the definition of ψ), $R_{\pi(i)} \dashrightarrow U^{[2k]}$. We therefore have

$$CR^{[\pi(i)-1]} \longrightarrow R_{\pi(i)} \dashrightarrow U^{[2k]} \longrightarrow cr^{[l,r]}$$

so $CR^{[\pi(i)-1]} \longrightarrow cr^{[l,r]}$. By part (b) of Proposition 6, this implies $\pi(i) - 1 \leq l$.

Proposition 7 implies $U^{[\psi(i)]} \dashrightarrow R_i$, so

$$cr^{[l,r]} \longrightarrow U^{[\psi(i)]} \dashrightarrow R_i \longrightarrow CR^{[i]}$$

This implies $cr^{[l,r]} \longrightarrow CR^{[i]}$, so part (a) of Proposition 6 implies $r \leq i$. We therefore have $\psi(i) - 1 \leq l \leq r \leq i$, so regularity of cr implies that $cr^{[l,r]}$ obtains a value $cr^{[j]}$ with $\psi(i) - 1 \leq j \leq i$. However, I already observed that all such values equal c, and $cr^{[l,r]}$ obtains the value $\neg c$. This is the required contradiction, completing the proof.

5 Conclusion

I have defined three classes of shared registers for asynchronous interprocess communication and provided algorithms for implementing one class in terms of a weaker class. For single-writer registers, the only unsolved problem is implementing a multireader atomic register. A solution probably exists, but it undoubtedly requires that a reader communicate with all other readers as well as with the writer. Also, more efficient implementations than Constructions 4 and 5 probably exist. For multivalued registers, Peterson's algorithm [11] combined with Construction 5 provides a more efficient implementation of a regular register than Construction 4, and a more efficient implementation of a single-reader atomic register than Construction 5. However, in this solution, Construction 4 is still needed to implement the regular register used in Construction 5.

I have not addressed the question of multiwriter shared registers. It is not clear what assumptions one should make about the effect of overlapping writes. The one case that is straightforward is that of an atomic multiwriter register—the kind of register traditionally assumed in shared-variable concurrent programs. This raises the problem of implementing a multiwriter atomic register from single-writer ones. An unpublished algorithm of Bard Bloom implements a two-writer atomic register using single-writer atomic registers.

In addition to studying shared registers, I have also developed a formalism for reasoning about concurrent systems that is not based upon atomic actions. Starting from a more general, relativistic viewpoint, I showed that one can, with no essential loss of generality, think in terms of starting and finishing times of operations. While starting and finishing times are intuitively more appealing, and can be useful in proving metatheorems about general systems, rigorous reasoning about specific algorithms is best done in the general formalism, using Axioms A1–A5. These axioms seem to contain the fundamental properties of temporal relations among operation executions that are needed to analyze concurrent algorithms.

References

[1] A. Mazurkiewicz. *Semantics of Concurrent Systems: A Modular Fixed Point Trace Approach.* Technical Report 84–19, Institute of Applied Mathematics and Computer Science, University of Leiden, 1984.

[2] W. Brauer, editor. *Net Theory and Applications.* Springer-Verlag, Berlin, 1980.

[3] P. J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):190–199, October 1971.

[4] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[5] Leslie Lamport. The mutual exclusion problem. To appear in *JACM*.

[6] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[7] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[8] Leslie Lamport. What it means for a concurrent program to satisfy a specification: why no one has specified priority. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN, New Orleans, January 1985.

[9] Peter E. Lauer, Michael W. Shields, and Eike Best. *Formal Theory of the Basic COSY Notation.* Technical Report TR143, Computing Laboratory, University of Newcastle upon Tyne, 1979.

[10] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, Berlin, 1980.

[11] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[12] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Symposium on the Foundations of Computer Science*, ACM, November 1977.

[13] Glynn Winskel. *Events in Computation.* PhD thesis, Edinburgh University, 1980.

Appendix

Proof of Proposition 1

It follows from (1) that, for any operation execution A in S, the relations $\longrightarrow$ and $\dashrightarrow$ are not changed by either of the following two changes to the global-time model, where $\delta > 0$:

1. Changing s_A to $s_A - \delta$ if, for all $B \in S$: $f_B < s_A$ implies $f_B < s_A - \delta$.
2. Changing f_A to $f_A + \delta$ if, for all $B \in S$: $f_A < s_B$ implies $f_A + \delta < s_A$.

Let T denote the set of numbers s_A and f_A for all A in S, and for any real t, let $S(t) = \{r \in T : r < t\}$ and $F(t) = \{r \in T : r > t\}$. M2 implies that for any t, $\max S(t) < t$ and $t < \min F(t)$.

For any A, if s_A equals s_B or f_B for some $B \neq A$, I can change s_A to $s_A - \delta$, where $0 < \delta < \epsilon$ is chosen so that $s_A - \delta > \max S(s_A)$. Similarly, if f_A equals s_B or f_B for some $B \neq A$, I can change f_A to $f_A + \delta$, where $0 < \delta < \epsilon$ and $f_A + \delta < \min F(s_A)$.

The details of the formal proof, which involves an inductive definition of s' and f' based upon the countability of S, is left to the reader.

Proof of Propositions 2 and 3

The "only if" part of Proposition 2 follows immediately from (1). To prove Proposition 3 and the "if" part of Proposition 2, I prove that, for every system execution $S, \longrightarrow, \dashrightarrow$, there exists a global-time model s, f such that for every $A, B \in S$:

- $A \longrightarrow B$ implies $f_A < s_B$
- $A \dashrightarrow B$ implies $s_A < f_B$

The relations $\xrightarrow{\prime}$ and $\overset{!}{\dashrightarrow}$ defined by this global-time model satisfy the requirements of Proposition 3. Moreover, if $S, \longrightarrow, \dashrightarrow$ satisfies A#, then $\overset{!}{\dashrightarrow}$ must equal $\dashrightarrow$, since if A# holds then $A \not\dashrightarrow B$ implies $B \longrightarrow A$, which implies $B \xrightarrow{\prime} A$, so $A \not\overset{\prime}{\dashrightarrow} B$, and $A \not\xrightarrow{\prime} B$ implies $B \dashrightarrow A$, which implies $B \overset{!}{\dashrightarrow} A$, so $A \not\xrightarrow{\prime} B$.

The following proposition is used in this proof and in a later one.

Proposition 10 *Let T be the set consisting of all elements of the form s_A and f_A for $A \in S$ (the elements of T are uninterpreted symbols, not necessarily real numbers), and let $\prec$ be the smallest transitively closed relation such that*

- *If $A \longrightarrow B$ then $f_A \prec s_B$.*
- *If $A \dashrightarrow B$ or $A = B$ then $s_A \prec f_B$.*

Then $\prec$ is an irreflexive partial ordering.

Proof: Define the relations $\xrightarrow{o}$, $\xrightarrow{s}$, and $\xrightarrow{d}$ on T as follows:

- For all A: $s_A \xrightarrow{o} f_A$.
- $f_A \xrightarrow{s} s_B$ if and only if $A \longrightarrow B$.
- $s_A \xrightarrow{d} f_B$ if and only if $A \dashrightarrow B$.

Let $\longrightarrow$ be the union of the three relations $\xrightarrow{o}$, $\xrightarrow{s}$, and $\xrightarrow{d}$, so $\prec$ is the transitive closure of $\longrightarrow$. It suffices to prove that $\longrightarrow$ is an acyclic relation.

The proof is by contradiction. Choose a shortest cycle formed by the $\longrightarrow$ relation. A cycle composed entirely of $\xrightarrow{o}$ and $\xrightarrow{s}$ relations would violate A1, so the cycle must contain a portion of the form:

$$f_A \xrightarrow{s} s_B \xrightarrow{d} f_C \xrightarrow{s} s_D$$

since $\xrightarrow{s}$ is the only relation from an f to an s and there are no s to s or f to f relations. I can apply A4 to deduce that $f_A \xrightarrow{s} s_D$, which contradicts our assumption that the cycle had minimal length, proving Proposition 10. ∎

Returning to the proof of Propositions 2 and 3, we see that $\prec$ is an irreflexive acyclic relation. Moreover, A5 implies that, for any $t \in T$, $t \prec s$ for all but a finite number of elements s. This, together with the countability of T, implies that $\prec$ can be completed to a total ordering $<$ such that there is an order-preserving isomorphism of T with a subset of the natural numbers. Identifying the elements of T with the corresponding natural numbers provides the desired global-time model.

Proof of Proposition 4

Let T be the set of all numbers s_A and f_A for $A \in S$, and let $\prec$ be the partial ordering on T defined as in Proposition 10 for the precedence relations $\xrightarrow{t}$

and $\dashrightarrow$, namely, the smallest partial order such that $A \xrightarrow{\ \prime\ } B$ implies $f_A \prec s_B$, and $A \dashrightarrow B$ or $A = B$ implies $s_A \prec f_B$. Observe that the following hold for all A and B in S:

(a) Either $s_A \prec f_B$ or $f_B \prec s_A$ (by A#).

(b) $f_A < s_B$ implies $f_A \prec s_B$ (by H3).

To prove the proposition, it suffices to construct s', f' such that[5] $s \leq s' \leq f' \leq f$ and for all A and B: $f_A \prec s_B$ implies $f'_A < s'_B$ and $s_A \prec f_B$ implies $s'_A < f'_B$.

Let s', f' be any global model satisfying

$$f'_A < s'_B \text{ implies } f_A \prec s_B \tag{5}$$

The pair of operation executions A, B is said to be *out of order for* s', f' if $f_A \prec s_B$ and $s'_B < f'_A$. It follows from (a) and (b) that, if there are no out-of-order pairs, then s', f' satisfies the conditions of the proposition.

I will construct s', f' inductively by constructing a sequence of nondegenerate models s^i, f^i with $s^i \leq s^{i+1} \leq f^{i+1} \leq f^i$ having s^0, f^0 equal to s, f and s', f' equal to their limit. This is done by first choosing the enumeration of all out-of-order pairs of s, f such that, for any subset of them, the minimal element is the one A, B having the smallest value of f_A and, among all such pairs A, B', the one having the largest value of s_B. It follows from M2 that such a minimal element exists for any nonempty set, so this defines an enumeration of the out-of-order pairs of s, f.

If A, B is the i^{th} out-of-order pair, then s^i, f^i will be defined to be the same as s^{i-1}, f^{i-1} except that $s^{i-1}_B < f^i_A < s^i_B < f^{i-1}_A$. This implies that the set of out-of-order pairs for s^i, f^i equals the set of out-of-order pairs for s^{i-1}, f^{i-1} minus the pair A, B. Moreover, it follows from A5 and (b) that any operation execution belongs to only a finite number of out-of-order pairs of s, f, so the limit s', f' of the models s^i, f^i exists, satisfies (5), and has no out-of-order pairs, proving the proposition.

For notational convenience, the construction of s^i, f^i from s^{i-1}, f^{i-1} is given for the case $i = 0$. So, I assume that s, f satisfies (b), which is the same as (5), and has a minimal out-of-order pair A, B. I construct s^1, f^1 by decreasing f_A and increasing s_B to get $f^1_A < s^1_B$, without creating any new out-of-order pairs. (The construction for any i is the same except with more superscripts.)

[5] I employ the usual notation that, for functions f and g with the same domain, $f \leq g$ if and only if $f(x) \leq g(x)$ for all x in their domain.

Let X be the operation execution with the largest value of s_X such that $s_X \prec f_A$; if there is no such X, let $s_X = -\infty$. It follows from (b) and the nondegeneracy of s, f that $s_X < f_A$. Observe that there is no C with s_C in the interval $(\max(s_X, s_B), f_A]$, since, by choice of s_X, this would imply $f_A \prec s_X$, which would contradict the maximality of s_B. Therefore, if I define f_A^1 to be $\max(s_X, s_B)^+$, then s, f^1 satisfies (5) and has the same set of out-of-order pairs as s, f, where t^+ denotes a value larger than t such that there is no value s_C or f_C in the interval $(t, t^+]$.

If $s_B > s_X$, so $f_A^1 = s_B^+$, then I can define s_B^1 to be $(f_A^1)^+$ and it is clear that s^1, f^1 also satisfies (5) and has the same set of out-of-order points as s, f^1 except that A, B is not out of order for s^1, f^1, so we are done.

Therefore, I need only consider the case $s_B < s_X$. (Since $s_X \prec f_A$, we must have $s_B \neq s_X$.) I claim that there is no f_C in the interval $[s_B, s_X]$. If there were, then (a) and (b) imply that $f_C \prec s_X$ and $s_B \prec f_C$, which, since $s_X \prec f_A$, would imply $s_B \prec f_A$, contrary to the assumption that A, B is out of order for s, f. Therefore, defining $s^{.5}$ to be the same as s except with $s_B^{.5} = s_X^+$, we see that $s^{.5}, f^1$ satisfies (5) and has the same set of out-of-order pairs as s, f^1. Replacing s by $s^{.5}$ and starting our argument again, we are in the case $s_X^{.5} < s_B^{.5}$ that was considered above. This completes the proof.

Proof of Proposition 5

If $\longrightarrow$ and $\dashrightarrow$ are any relations in a set S, let the *completion* of $\longrightarrow$ and $\dashrightarrow$ be the relations $\xrightarrow{!}$ and $\dashrightarrow^{!}$, where $\xrightarrow{!}$ is the smallest transitively closed extension of $\longrightarrow$ such that $A \xrightarrow{!} B \dashrightarrow C \xrightarrow{!} D$ implies $A \xrightarrow{!} D$, and $\dashrightarrow^{!}$ is the union of $\dashrightarrow$ and $\xrightarrow{!}$. Thus, $A \xrightarrow{!} B$ if and only if there exists a chain

$$A = A_1 \Longrightarrow \cdots \Longrightarrow A_n = B$$

where $\Longrightarrow$ denotes either $\longrightarrow$ or $\longrightarrow C \dashrightarrow D \longrightarrow$ for some C and D.

Proposition 11 *If $\longrightarrow$ satisfies A5; $\xrightarrow{!}, \dashrightarrow^{!}$ is the completion of $\longrightarrow, \dashrightarrow$; and $\xrightarrow{!}$ is acyclic; then $S, \xrightarrow{!}, \dashrightarrow^{!}$ is a system execution.*

Proof: I must show that $S, \xrightarrow{!}, \dashrightarrow^{!}$ satisfies A1–A5. The only nonobvious part is, in the proof of A2, showing that, if $A \xrightarrow{!} B$, then $B \dashrightarrow^{!} A$. However, as observed above, this follows from A1 and A4. ∎

To prove Proposition 5, let $\xrightarrow{o}$ be the union of the relations $\xrightarrow{*}$ and $\xrightarrow{\mathcal{H}}$, and let $\dashrightarrow^{o}$ be the union of $\dashrightarrow^{\mathcal{H}}$ and the restriction of $\xrightarrow{*}$ to $\mathcal{T}$. Note

that the restriction of $\xrightarrow{o}$ to $\mathcal{H}$ equals $\xrightarrow{\mathcal{H}}$ (by H3). I define $\xrightarrow{\mathcal{HT}}, \dashrightarrow^{\mathcal{HT}}$ to be the completion of $\xrightarrow{o}, \dashrightarrow^{o}$.

I claim that, to prove Proposition 5, it suffices to show that $\xrightarrow{\mathcal{HT}}$ is acyclic and the restrictions of $\xrightarrow{\mathcal{HT}}$ and $\dashrightarrow^{\mathcal{HT}}$ to $\mathcal{H}$ equal $\xrightarrow{\mathcal{H}}$ and $\dashrightarrow^{\mathcal{H}}$. Proposition 11 then implies that $\mathcal{H} \cup \mathcal{T}, \xrightarrow{\mathcal{HT}}, \dashrightarrow^{\mathcal{HT}}$ is a system execution, which is easily seen to be implemented by $\mathcal{S} \cup \mathcal{T}, \longrightarrow, \dashrightarrow$. (The definition of $\xrightarrow{\mathcal{HT}}$ and $\dashrightarrow^{\mathcal{HT}}$ implies that their restrictions to $\mathcal{T}$ are extensions of $\xrightarrow{\bullet}$ and $\dashrightarrow^{\bullet}$.)

Moreover, I claim that it suffices to prove that the restriction of $\xrightarrow{\mathcal{HT}}$ to $\mathcal{H}$ equals $\xrightarrow{\mathcal{H}}$. It follows immediately from the definition of $\dashrightarrow^{\mathcal{HT}}$ and A2 that, if the restriction of $\xrightarrow{\mathcal{HT}}$ equals $\xrightarrow{\mathcal{H}}$, then the restriction of $\dashrightarrow^{\mathcal{HT}}$ to $\mathcal{H}$ must equal $\dashrightarrow^{\mathcal{H}}$. Furthermore, the definition of the completion and the acyclicity of $\xrightarrow{\bullet}$ imply that, any cycle of $\xrightarrow{\mathcal{HT}}$ relations must include an element of $\mathcal{H}$, so $A \xrightarrow{\mathcal{HT}} A$ must hold for some $A \in \mathcal{H}$. If the restriction of $\xrightarrow{\mathcal{HT}}$ to $\mathcal{H}$ equals $\xrightarrow{\mathcal{H}}$, then the acyclicity of $\xrightarrow{\mathcal{HT}}$ follows from the acyclicity of $\xrightarrow{\mathcal{H}}$. Thus, it suffices to prove that, if $A \xrightarrow{\mathcal{HT}} B$, then $A \xrightarrow{\mathcal{H}} B$.

By definition of $\xrightarrow{\mathcal{HT}}$, if $A \xrightarrow{\mathcal{HT}} B$ then there exists a chain $A = A_1 \Longrightarrow \cdots \Longrightarrow A_n = B$, where $\Longrightarrow$ denotes either $\xrightarrow{o}$ or $\xrightarrow{o} C \dashrightarrow^{\bullet} D \xrightarrow{o}$. Note that, f A_i and A_{i+1} are both in $\mathcal{H}$, then $A_i \Longrightarrow A_{i+1}$ implies that $A_i \xrightarrow{\mathcal{H}} A_{i+1}$, and, if they are both in $\mathcal{T}$, then $A_i \Longrightarrow A_{i+1}$ implies that $A_i \xrightarrow{\bullet} A_{i+1}$. Therefore, it suffices to show that any such chain that is of minimal length has length one.

If three consecutive elements A_i, A_{i+1}, and A_{i+2} in this chain are either all in $\mathcal{H}$ or all in $\mathcal{T}$, by the transitivity of $\xrightarrow{\mathcal{H}}$ and $\xrightarrow{\bullet}$ it follows that $A_i \Longrightarrow A_{i+2}$. Therefore, in a minimal-length chain, A_i must be in $\mathcal{H}$ if i is odd and in $\mathcal{T}$ if i is even. If $n > 0$, then we have $A_1 \Longrightarrow A_2 \Longrightarrow A_3$, with A_1 and A_3 in $\mathcal{H}$ and A_2 in $\mathcal{T}$. A $\xrightarrow{o}$ relation between an element of $\mathcal{H}$ and an element of $\mathcal{T}$ must be a $\xrightarrow{\bullet}$ relation. Considering the two possible cases for each $\Longrightarrow$ relation, using A1 and A4 for the relations $\xrightarrow{\bullet}$ and $\dashrightarrow^{\bullet}$, it follows from $A_1 \Longrightarrow A_2 \Longrightarrow A_3$ that $A_1 \xrightarrow{\bullet} A_2 \xrightarrow{\bullet} A_3$, so $A_1 \Longrightarrow A_3$. This contradicts the assumption of the minimality of n, proving that $n = 1$ and $A \xrightarrow{\mathcal{H}} B$, which completes the proof of the proposition.

Proof of Propositions 6 and 7

Parts (a) and (b) of Proposition 6 are an immediate consequence of Definition 4. To prove part (c), observe that this definition implies $V^{[i]} \dashrightarrow v^{[i,j]}$.

The result is immediate if $j = 0$. If $j > 0$, then $V^{[j-1]} \longrightarrow V^{[j]}$. Combining these two relations with the hypothesis, we have

$$V^{[j-1]} \longrightarrow V^{[j]} \dashrightarrow v^{[i,j]} \longrightarrow v^{[i',j']}$$

Axiom A4 implies that $V^{[j-1]} \longrightarrow v^{[i',j']}$, which, by A2, implies $v^{[i',j']} \not\dashrightarrow V^{[j-1]}$. This finishes the proof of Proposition 6.

To prove part (a) of Proposition 7, observe that it follows immediately from Definition 4 that $V^{[k]} \dashrightarrow R$ implies $k \leq j$. Conversely, I assume $k \leq j$ and show this implies $V^{[k]} \dashrightarrow R$. Since $V^{[j]} \dashrightarrow R$, the desired conclusion is immediate if $k = j$. If $k < j$, then $V^{[k]} \longrightarrow V^{[j]}$, and it follows from A3.

For part (b), Definition 7 implies that, if $i < k'$, then $R \dashrightarrow V^{[k']}$. Letting $k' = k + 1$, this shows that, if $i \leq k$, then $R \dashrightarrow V^{[k+1]}$. Conversely, suppose $R \dashrightarrow V^{[k+1]}$, then $k + 1 \neq i$. If $k + 1 < i$, then $V^{[k+1]} \longrightarrow V^{[i]}$, so A3 would imply $R \dashrightarrow V^{[i]}$ contrary to Definition 4. Hence, we must have $i < k + 1$ so $i \leq k$, completing the proof of Proposition 7.

Proof of Propositions 8 and 9

Apply Proposition 3 to extend the given $\longrightarrow$ and $\dashrightarrow$ relations so they satisfy A#. It follows from B1 that this extension does not add any new precedence relations between reads and writes. A read sees $v^{[i,j]}$, as defined by these new relations, if and only if it sees $v^{[i,j]}$ in the original system execution. Hence, the new system execution, which satisfies A#, satisfies the hypotheses of the appropriate proposition. Applying Proposition 2, I can therefore assume a nondegenerate global-time model for the system execution.

For the proof of Proposition 9, let ϕ be the assumed function. For the proof of Proposition 8, ϕ is defined as follows. If R is a read that sees $v^{[i,j]}$, for a safe register define $\phi(R)$ to equal j, and for a regular register define it to be a value satisfying conditions 1 and 2 in the hypothesis of Proposition 9. (B4 implies that such a definition is possible.)

I first show that $S, \longrightarrow, \dashrightarrow$ (which I am assuming to have a nondegenerate global-time model) trivially implements a system execution in which reads are instantaneous, which is all that is required to prove Proposition 8. Given the nondegenerate global-time model s, f for $S, \longrightarrow, \dashrightarrow$, it suffices to find a global-time model s', f' with $s \leq s' \leq f' \leq f$ in which all reads are instantaneous, such that B1–B4 hold for the system execution defined by s', f'.

For notational convenience, let s_i and f_i denote $s_{V^{[i]}}$ and $f_{V^{[i]}}$, respectively. Let s', f' be the same as s, f except that, for a read R, define s'_R to

equal the maximum of the following three quantities:

- s_R
- $(s_{\phi(R)})^+$
- $\max\{s_{R'} : \phi(R') < \phi(R) \text{ and } s_{R'} < f_R\}^+$

and define f'_R to equal $(s'_R)^+$. When the appropriate careful definition of t^+ is given, this results in a nondegenerate global-time model in which every read is instantaneous. I must check that, for any read R: $s_R \leq s'_R \leq f'_R \leq f_R$, B1–B3 remain satisfied, and B4 remains satisfies when v is regular.

It is immediate by the definition of s'_R that $s_R \leq s'_R$. Since $f'_R = (s'_R)^+$, to establish the remaining inequalities I need to show that $f'_R < f_R$. If R sees $v^{[i,j]}$, then, by Definition 4, $s_j < f_R$ (the strict inequality comes from nondegeneracy), and, since $\phi(R) \leq j$, $s_{\phi(R)} < f_R$. The required inequality now follows easily from the definition of s'_R.

I must now show that B1–B3 and, if v is regular, B4 hold for the new precedence relations. B1 and B2 are trivial. For B3 and B4, consider what a read sees in the new system execution if it sees $v^{[i,j]}$ in the original one. There are three cases:

1. If $f_{\phi(R)} < s_R$ then
   (a) if $s_R < s_{\phi(R)+1}$ then R sees $v^{[\phi(R),\phi(R)]}$
   (b) if $s_{\phi(R)+1} < s_R$ then R sees $v^{[\phi(R),\phi(R)+1]}$
2. If $s_R < f_{\phi(R)}$ then R sees $v^{[\phi(R)-1,\phi(R)]}$.

Moreover, it is immediate from Definition 4 that case 1(b) is impossible if $\phi(R) = j$, which is the case when v is assumed to be only safe. This definition also implies that $f_j < s_R$ if and only if $i = j$. Thus, when v is only safe, R sees $v^{[i,i]}$ in the new system execution if and only if it does in the old, proving B3. For the case when v is regular, B3 and B4 follow immediately from the fact that R returns the value $v^{[\phi(R)]}$. This finishes the proof of Proposition 8.

To complete the proof of Proposition 9, I first show that, if $\phi(R) < \phi(S)$ for reads R and S, then $f'_R < s'_S$. The third hypothesis about ϕ implies that, if $\phi(R) < \phi(S)$, then $s_R < f_S$. By the definition of s'_S, this implies that s'_S is greater than each of the three quantities of which s'_R is the maximum, so $s'_R < s'_S$. Since reads are instantaneous with respect to s', f', this implies $f'_R < s'_R$.

I must construct a new global-time model s'', f'', in which writes are also instantaneous and B1–B3 are still satisfied, so that s'', f'' is the same as s', f' except for writes, and for any write $V^{[k]}$: $s'_k \leq s''_k \leq f''_k \leq f'_k$. (Note that B5 follows from the fact that reads and writes are instantaneous, and B4 follows from B3 and B5.)

Let s''_k be the maximum of the two quantities s'_k and $\max\{f'_R : \phi(R) = k-1\}^+$, and let f''_k be $(s''_k)^+$. Since $v^{[\phi(R)]}$ is one of the values "seen" by R in the system execution defined by s', f', if $\phi(R) = k-1$ then $s'_R < f'_k$, which implies that $s''_k < f'_k$. We therefore have $s' \leq s'' \leq f'' \leq f'$, and reads and writes are both instantaneous in s'', f''. Again, B1 and B2 are trivial, so I need only prove B3.

Since reads and writes are instantaneous, B5 holds—a read R sees $v^{[i,i]}$; I must show that $i = \phi(R)$. The definition of s'' implies that $f''_R = f'_R < s''_{\phi(R)+1}$. I must therefore show that $s''_{\phi(R)} < s'_R$. In the global-time model s', f', the read R "sees the value" $v^{[\phi(R)]}$, so $s'_{\phi(R)} < s'_R$. By definition of s'', we can have $s''_{\phi(R)} > s'_R$ only if there exists some R' with $\phi(R') < \phi(R)$ and $f_{R'} > s'_R$. However, I showed above that $R' < R$ implies $f'_{R'} < s'_R$, which completes the proof.

END

FILMED

8-85

DTIC